GUI - lecture notes

with example programs from archive ***java_ examples***

Tomasz R. Werner

Warsaw, June 5, 2021 13:59

Contents

Section 1

Abstract classes, interfaces, inner classes and lambdas

1.1 Abstract classes

Abstract classes are those which contain at least one abstract method, i.e., a method only declared but not defined. Both abstract classes and methods must be marked as such with the keyword **abstract**. It is not possible to create an object of an abstract class (as it is not fully implemented) — their *raison d'être* is to be extended. Extending (inheriting) class must provide definitions of methods declared but not defined in its abstract superclass, otherwise it will still be abstract itself. As a matter of fact, an abstract class *may* have all its methods implemented and still be declared as **abstract** — then it will be still impossible to create its objects and its only purpose is to serve as a base class for other classes.

Abstract classes provide a common interface (collection of methods) to a set of classes which can implement these methods in different ways. What is important is that we can declare references to abstract types while objects pointed to by these references are all of derived types (they cannot be objects of the base, abstract, class because such object cannot be even created).

Let us consider an example of an abstract class **Figure**, representing geometrical figures. Of course, if all we know is that something is a figure, we cannot say much about its area or perimeter: for this we should know its type more precisely. Therefore **getArea** and **getPerimeter** are declared abstract:

Listing 1 EPZ-AbsFig/Figure.java

```java
abstract class Figure {

    abstract public double getArea();
    abstract public double getPerimeter();

    static public Figure getFigMaxArea(Figure[] figs) {
        double maxarea = 0;
        Figure maxfig  = null;
        for (Figure f : figs) {
            double area = f.getArea();
            if (area > maxarea) {
                maxarea = area;
                maxfig  = f;
            }
        }
        return maxfig;
    }

    @Override
    public String toString() {
        return " area: "
              + String.format("%6.3f",getArea())
              + "; perimeter: "
              + String.format("%6.3f",getPerimeter());
    }
}
```

Class **Circle** inherits **Figure** and provides definitions of the abstract methods. It also overrides **toString** — note, how its implementation uses that from the base class (by using **super**).

Listing 2 EPZ-AbsFig/Circle.java

```
public class Circle extends Figure {
    private double r;

    public Circle(double r) {
        this.r = r;
    }

    @Override
    public double getArea() {
        return Math.PI*r*r;
    }
    @Override
    public double getPerimeter() {
        return 2*Math.PI*r;
    }
    @Override
    public String toString() {
        return "Circle    (   r=" + r + "   )"
                + super.toString();
    }
}
```

as does **Rectangle**

Listing 3 EPZ-AbsFig/Rectangle.java

```
public class Rectangle extends Figure {
    private double a, b;

    public Rectangle(double a, double b) {
        this.a = a;
        this.b = b;
    }

    @Override
    public double getArea() {
        return a*b;
```

```java
    }

    @Override
    public double getPerimeter() {
        return 2*(a+b);
    }

    @Override
    public String toString() {
        return "Rectangle (a=" + a + " b=" + b + ")"
              + super.toString();
    }
}
```

and we can now check our classes in

Listing 4 EPZ-AbsFig/Main.java

```java
import java.util.Locale;

public class Main {
    public static void main(String[] args) {
          // to have decimal point instead of a comma...
        Locale.setDefault(Locale.US);

        Figure[] figs = {
            new Circle(2),      new Rectangle(9,1),
            new Rectangle(4,3), new Circle(4)
        };

        Figure fig = Figure.getFigMaxArea(figs);
        System.out.println("\nLargest area: \n" + fig);
    }
}
```

1.2 Interfaces

Interfaces are, in a sense, “pure” abstract classes — they only declare one or more methods, but do not implement them (but see below). All methods are, by definition, **public**, even if not declared as such. Also, unimplemented methods are *not* declared as **abstract** (although they are abstract).
Interfaces can also *define*:

- final, static constants;
- default methods, marked with the keyword **default**;
- *private* methods, used internally by its default methods — these private methods, of course, are not visible for the user and do not belong to the ‘contract’.
- *static* methods — which can be public and are accessible for the users.

Interfaces with *only one* abstract method (but, perhaps, with some default methods and static member functions already implemented) are called **functional interfaces** and play a special rôle in Java (more about it later — see 7, p. 80).

What's important is the fact that any class can *extend* (directly inherit from) only one class, but may *implement* **any number of interfaces**.

Interfaces define a "contract" — if we know that a class implements a given interface, we know that all its abstract methods must have been implemented somehow and hence it is safe to call these methods on object of such a class. We can also declare references of interface type — they can refer to objects of *any* classes implementing the interface, even if their types belong to completely separate subtrees of the class hierarchy.

Let us consider the following example. We define an interface, **IMyStack**, which defines a 'contract' of a stack. What is a stack? It is something that has **pop**, **push** and **empty** methods. We can then create concrete classes that implement this interface in different ways — for example using an array or using the singly-linked list (**MyStackArr** and **MyStackList**, respectively, in the example below). Note the function **testStack**. Its single argument is declared as 'something' of type **IMyStack** — there will be no objects of this type because it is not a concrete class but an interface. However, this declaration means 'anything what implements **IMyStack** will be accepted'. We can then safely call **push**, **pop** and **empty** on the object passed to the function because we know that they have to be implemented somehow.

Listing 5 ELJ-InterStack/MyStacks.java

```java
public class MyStacks {
    public static void main(String[] args) {
        testStack(new MyStackArr(10));
        testStack(new MyStackList());
    }

    public static void testStack(IMyStack stack) {
        stack.push(5);
        stack.push(4);
        stack.push(3);
        stack.push(2);
        while (!stack.empty()) {
            System.out.print(stack.pop() + " ");
        }
        System.out.println();
    }
}


interface IMyStack {
    int pop();          // public automatically
    void push(int i);
    boolean empty();
}

class MyStackArr implements IMyStack {
    int[] arr;
    int   top;
    public MyStackArr(int size) {
        arr = new int[size];
        top = 0;
    }
    public int pop() {
        return arr[--top];
    }
    public void push(int i) {
        arr[top++] = i;
    }
    public boolean empty() {
        return top == 0;
    }
}

class MyStackList implements IMyStack {
    private static class Node {
        int data;
        Node next;
        Node(int d, Node n) {
            data = d;
            next = n;
        }
        Node(int d) {
            this(d, null);
        }
    }
    private Node head = null;
    public int pop() {
        int d = head.data;
        head = head.next;
        return d;
    }
    public void push(int d) {
        head = new Node(d, head);
    }
    public boolean empty() {
        return head == null;
    }
}
```

Let us now consider another example. We define an interface that has only one abstract method: **lt** (*less than*). All other methods (**gt** — *greater than*, **ge** — *greater or equal than*, etc.) have default implementation expressed, directly or indirectly, by this one abstract method. Therefore, to implement this interface we only have to implement **lt** — all other methods will then work automatically!

Listing 6 ELS-DefIntSimple/MyCompar.java

```java
public interface MyCompar {
      // abstract
    boolean lt(int lhs, int rhs);

      // implemented directly in terms of the abstract
    default boolean gt(int lhs, int rhs) {
        return lt(rhs,lhs);
    }
    default boolean ge(int lhs, int rhs) {
        return !lt(lhs,rhs);
    }
    default boolean le(int lhs, int rhs) {
        return !lt(rhs,lhs);
    }
    default boolean eq(int lhs, int rhs) {
        return !lt(lhs,rhs) && !lt(rhs,lhs);
    }
      // implemented indirectly in terms of the abstract
    default boolean ne(int lhs, int rhs) {
        return !eq(lhs,rhs);
    }
}
```

Then we define three classes implementing this interface: one compares integers by their values

Listing 7 ELS-DefIntSimple/CompVal.java

```java
public class CompVal implements MyCompar {
    @Override
    public boolean lt(int lhs, int rhs) {
        return lhs < rhs;
    }
}
```

the second by sum of digits

Listing 8 ELS-DefIntSimple/CompDigits.java

```java
public class CompDigits implements MyCompar {
    @Override
    public boolean lt(int lhs, int rhs) {
        return sumOfDigs(lhs) < sumOfDigs(rhs);
    }
    private static int sumOfDigs(int n) {
        int sum = 0;
        n = n < 0 ? -n : n;
        while (n != 0) {
            sum += n % 10;
            n /= 10;
        }
        return sum;
    }
}
```

and the third by value, but reversed

Listing 9 ELS-DefIntSimple/CompValRev.java

```
public class CompValRev implements MyCompar {
    @Override
    public boolean lt(int lhs, int rhs) {
        return lhs > rhs;
    }
}
```

Now in **Main** we can use all three implementations; each of them has full set of all methods implemented although they override only one method:

Listing 10 ELS-DefIntSimple/Main.java

```
public class Main {
    public static void main (String[] args) {

        MyCompar cmpVal = new CompVal();
        MyCompar cmpSum = new CompDigits();
        MyCompar cmpVaR = new CompValRev();

        compare("cmpVal - BY VALUE",cmpVal,
                10,2, 3,12, 5,22);
        compare("cmpSum - BY SUM OF DIGITS",cmpSum,
                10,2, 3,12, 5,22);
        compare("cmpVaR - BY VALUE REVERSED",cmpVaR,
                10,2, 3,12, 5,22);
    }

    private static void compare(String message,
                    MyCompar cmp, int... pairs) {
        System.out.println("\n========= " + message);
        for (int k = 0; k < pairs.length; k += 2) {
            int a = pairs[k], b = pairs[k+1];
            System.out.println("** (" + a + "," + b + "): "+
                    "lt->" + cmp.lt(a,b) + ", " +
                    "le->" + cmp.le(a,b) + "\n" +
                "    gt->" + cmp.gt(a,b) + ", " +
                    "ge->" + cmp.ge(a,b) + ", " +
                    "eq->" + cmp.eq(a,b) + ", " +
                    "ne->" + cmp.ne(a,b) );
        }
    }
}
```

The program prints

```
========= cmpVal - BY VALUE
** (10,2): lt->false, le->false
    gt->true, ge->true, eq->false, ne->true
** (3,12): lt->true, le->true
    gt->false, ge->false, eq->false, ne->true
** (5,22): lt->true, le->true
    gt->false, ge->false, eq->false, ne->true

========= cmpSum - BY SUM OF DIGITS
** (10,2): lt->true, le->true
    gt->false, ge->false, eq->false, ne->true
** (3,12): lt->false, le->true
    gt->false, ge->true, eq->true, ne->false
** (5,22): lt->false, le->false
    gt->true, ge->true, eq->false, ne->true

========= cmpVaR - BY VALUE REVERSED
** (10,2): lt->true, le->true
    gt->false, ge->false, eq->false, ne->true
** (3,12): lt->false, le->false
    gt->true, ge->true, eq->false, ne->true
** (5,22): lt->false, le->false
    gt->true, ge->true, eq->false, ne->true
```

Let us now consider an example of a functional interface with abstract method. In the program below, we define **Fun** interface which declares one abstract method (**apply**) but is additionally equipped with a static function (**transformArray**) taking one array of doubles and returning another whose elements are the results of applying a function to elemensts of the input array. The function takes, as its second argument, the reference to an object of any class implementing the **Fun** interface and therefore providing a definition of the **apply** method:

Listing 11 ELL-InterFun/InterFun.java

```
public class Main {
    public static void main (String[] args) {

        MyCompar cmpVal = new CompVal();
        MyCompar cmpSum = new CompDigits();
        MyCompar cmpVaR = new CompValRev();

        compare("cmpVal - BY VALUE",cmpVal,
                10,2, 3,12, 5,22);
        compare("cmpSum - BY SUM OF DIGITS",cmpSum,
                10,2, 3,12, 5,22);
        compare("cmpVaR - BY VALUE REVERSED",cmpVaR,
                10,2, 3,12, 5,22);
    }

    private static void compare(String message,
                    MyCompar cmp, int... pairs) {
        System.out.println("\n========= " + message);
        for (int k = 0; k < pairs.length; k += 2) {
            int a = pairs[k], b = pairs[k+1];
            System.out.println("** (" + a + "," + b + "): "+
                    "lt->" + cmp.lt(a,b) + ", " +
                    "le->" + cmp.le(a,b) + "\n" +
                "    gt->" + cmp.gt(a,b) + ", " +
                    "ge->" + cmp.ge(a,b) + ", " +
                    "eq->" + cmp.eq(a,b) + ", " +
                    "ne->" + cmp.ne(a,b) );
        }
    }
}
```

Here, we first apply the **sin** function to an array of doubles, and then we apply multiplication by 2 to the resulting array. The program prints

```
[-0.9999999999999999, 0.9999999999999999, -0.9999999999999994]
```

Many important interfaces are already defined in the standard library. For example, Comparable declares one abstract method

```
int compareTo(Object)
```

Given objects a and b, a.compareTo(b) returns something negative if a is "smaller" than b, something positive if b is smaller, and 0 if they are considered equal. When declaring that a class implements this interface for objects of type T, we should always indicate this type (in angle brackets) — then we can declare the type of the argument as T and not Object (and no casting is required). Classes implementing Comparable are said to be equipped with **natural order**. .

In the (abstract) class Figure below , we declare and define **compareTo** (remember that all methods declared in an interface are by definition public, so overriding them we must not forget about public) specifically for Figures and therefore we declare the class as implementing Comparable<Figure>:

Listing 12 EQA-AbstractFigs/Figure.java

```
abstract class Figure implements Comparable<Figure> {

    abstract public double getArea();
    abstract public double getPerimeter();

    static public Figure getFigMaxArea(Figure[] figs) {
        double maxarea = 0;
        Figure maxfig  = null;
        for (Figure f : figs) {
            double area = f.getArea();
            if (area > maxarea) {
                maxarea = area;
                maxfig  = f;
            }
        }
        return maxfig;
    }

    @Override
    public String toString() {
        return " area: "
              + String.format("%6.3f",getArea())
              + "; perimeter: "
              + String.format("%6.3f",getPerimeter());
    }

    @Override
    public int compareTo(Figure f) {
        double diff = getPerimeter() - f.getPerimeter();
        if      (diff < 0) return -1;
        else if (diff > 0) return +1;
        else               return 0;
    }
}
```

Then we can define implementing classes: **Circle**

Listing 13 EQA-AbstractFigs/Circle.java

```
public class Circle extends Figure {
    private double r;

    public Circle(double r) {
        this.r = r;
    }

    @Override
    public double getArea() {
        return Math.PI*r*r;
    }
    @Override
    public double getPerimeter() {
        return 2*Math.PI*r;
    }
    @Override
```

```java
    public String toString() {
        return "Circle    (   r=" + r + "   )"
                + super.toString();
    }
}
```

and **Rectangle**

Listing 14 EQA-AbstractFigs/Rectangle.java

```java
public class Rectangle extends Figure {
    private double a, b;

    public Rectangle(double a, double b) {
        this.a = a;
        this.b = b;
    }

    @Override
    public double getArea() {
        return a*b;
    }
    @Override
    public double getPerimeter() {
        return 2*(a+b);
    }

    @Override
    public String toString() {
        return "Rectangle (a=" + a + " b=" + b + ")"
              + super.toString();
    }
}
```

Then in **main** we can use **sort** and Java will know how to compare figures: it will just call **compareTo** (it knows it is there, because the compiler can see that **Figure** implements **Comparable**, otherwise the program would not even compile):

Listing 15 EQA-AbstractFigs/Main.java

```java
import java.util.Arrays;
import java.util.Locale;

public class Main {
    public static void main(String[] args) {
          // to have decimal point instead of a comma...
        Locale.setDefault(Locale.US);

        Figure[] figs = {
            new Circle(2),      new Rectangle(9,1),
            new Rectangle(4,3), new Circle(4)
        };

        Figure fig = Figure.getFigMaxArea(figs);
        System.out.println("\nLargest area: \n" + fig);

        Arrays.sort(figs);
        System.out.println("\nSorted by circumference:");
        for (Figure f : figs)
            System.out.println(f);
    }
}
```

The program prints

```
Largest area:
Circle    (    r=4.0    ) area: 50.265; perimeter: 25.133

Sorted by circumference:
Circle    (    r=2.0    ) area: 12.566; perimeter: 12.566
Rectangle (a=4.0 b=3.0) area: 12.000; perimeter: 14.000
Rectangle (a=9.0 b=1.0) area:  9.000; perimeter: 20.000
Circle    (    r=4.0    ) area: 50.265; perimeter: 25.133
```

Any class can implement only one natural order. However it may happen that we want to use (e.g., for sorting) different criteria. We then can create an object which will be used as a comparator even if a natural order exist: it will be an object of a type implementing **Comparator** with only one abstract method

```java
int compare(Object,Object)
```

Then, if `ob` is an object of this class and `a` and `b` are to be compared, `ob.compare(a,b)` should return something negative if `a` is “smaller” than `b`, something positive if `b` is smaller, and 0 if they are considered equal. As with **Comparable**, when declaring a class implementing **Comparator**, we should always indicate the type, call it **T**, of objects it is supposed to be able to compare (in angle brackets) — then we can declare the type of the arguments of **compare** as **T** and not **Object** (and no casting is required).

Let us consider an example: class **Person** has a natural order (as it implements **Comparable**)

Listing 16 ELP-Comps2/Person.java

```java
public class Person implements Comparable<Person> {

    final static int currentYear =
            java.util.Calendar.getInstance().
                get(java.util.Calendar.YEAR);

    String name;
    int    birthYear;
    int    height;

    Person(String n, int y, int h) {
        name      = n;
        birthYear = y;
        height    = h;
    }

    /**
     * natural order: by name, then age, then height
     */
    @Override
    public int compareTo(Person o) {
        int k = name.compareToIgnoreCase(o.name);
        if ( k != 0 ) return k;
        k = o.birthYear - birthYear;
        if ( k != 0 ) return k;
        return height - o.height;
    }

    public String toString() {
        return name + "(" + (currentYear-birthYear) +
                                  "/" + height + ")";
    }
}
```

In order to be able to compare persons in different ways, not necessarily determined by the natural order, we define two different classes representing comparators of **Persons**

Listing 17 ELP-Comps2/Comparators.java

```java
import java.util.Comparator;

// package classes (not public)

/**
 * Comparator 1: by height, then age , then name
 */
class Comp1 implements Comparator<Person> {
    @Override
    public int compare(Person o1, Person o2) {
        int k = o1.height - o2.height;
        if ( k != 0 ) return k;
        k = o2.birthYear - o1.birthYear;
        if ( k != 0 ) return k;
        return o1.name.compareToIgnoreCase(o2.name);
    }
}

```

```
/**
 * Comparator 2: by age, then by name, then by height
 */
class Comp2 implements Comparator<Person> {
    @Override
    public int compare(Person o1, Person o2) {
        int k = o2.birthYear - o1.birthYear;
        if ( k != 0 ) return k;
        k = o1.name.compareToIgnoreCase(o2.name);
        if ( k != 0 ) return k;
        return o1.height-o2.height;
    }
}
```

which now can be used, for example, to sort arrays or lists of **Persons**:

Listing 18 ELP-Comps2/Main.java

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        new Main();
    }

    Main() {
        List<Person> list = new ArrayList<Person>();
        list.add(new Person("K",1980,165));
        list.add(new Person("B",1986,171));
        list.add(new Person("K",1980,168));
        list.add(new Person("H",1980,171));
        list.add(new Person("M",1980,171));
        list.add(new Person("K",1980,169));
        list.add(new Person("B",1979,171));
        list.add(new Person("G",1975,171));

          // natural
        Collections.sort(list);
        writeL(list, "Natural: name, age, height");

          // comparator Comp1
        Collections.sort(list, new Comp1());
        writeL(list, "Comp1:   height, age, name");

          // comparator Comp2
        Comparator<Person> comp2 = new Comp2();
        Collections.sort(list, comp2);
        writeL(list, "Comp2:   age, name, height");

          // anonymous comparator
        Collections.sort(list, new Comparator<Person>() {
            @Override
            public int compare(Person p, Person q) {
                int k = p.name.compareToIgnoreCase(q.name);
                if ( k != 0 ) return k;
                k = p.height - q.height;
                if ( k != 0 ) return k;
                return q.birthYear - p.birthYear;
            }
        });
        writeL(list, "Anonym:  name, height, age");

          // lambda
        Collections.sort(list, (f,s) -> f.height-s.height);
        writeL(list, "Lambda:  name, height, age");
    }

    static void writeL(List<Person> list, String header) {
        System.out.println('\n'+header);
        for (Person p : list) System.out.print(p+" ");
        System.out.println();
    }
}
```

The program prints

```
Natural: name, age, height
B(33/171) B(40/171) G(44/171) H(39/171)
K(39/165) K(39/168) K(39/169) M(39/171)

Comp1:   height, age, name
K(39/165) K(39/168) K(39/169) B(33/171)
H(39/171) M(39/171) B(40/171) G(44/171)

Comp2:   age, name, height
B(33/171) H(39/171) K(39/165) K(39/168)
K(39/169) M(39/171) B(40/171) G(44/171)

Anonym:  name, height, age
B(33/171) B(40/171) G(44/171) H(39/171)
K(39/165) K(39/168) K(39/169) M(39/171)

Lambda:  name, height, age
K(39/165) K(39/168) K(39/169) B(33/171)
B(40/171) G(44/171) H(39/171) M(39/171)
```

Let us consider another, but similar, example. We again create a class representing persons (equipped with a natural order)

Listing 19 ELM-Comps1/Person.java

```java
public class Person implements Comparable<Person> {

    private String name;
    private int    year;

    public Person(String name, int year) {
        this.name = name;
        this.year = year;
    }

    @Override
    public int compareTo(Person other) {
        int diff = year - other.year;
        if (diff != 0) return diff;
        else           return name.compareTo(other.name);
    }

    public String getName() { return name; }
    public int getYear()    { return year; }

    @Override
    public String toString() {
        return name + "(" + year + ")";
    }

    static void show(Person[] persons, String message) {
        System.out.println(message);
        for (Person person : persons)
            System.out.print(person + " ");
        System.out.println("\n");
    }
}
```

Now, we create only one class, **CompPerson**, objects of which can be used as comparators of **Person**s. The class contains one field, set by the constructor. In the example below it is an enumerator, but equally well it could have been an integer. When creating an object of this class, we will pass to the constructor information about the way we want persons to be compared — therefore, our class is in a way ‘configurable’

Listing 20 ELM-Comps1/CompPerson.java

```java
import java.util.Comparator;

public class CompPerson implements Comparator<Person> {

    public static enum Comp { BY_NAME,    BY_YEAR,
                              BY_NAMERev, BY_YEARRev };
    private Comp comp;

    public CompPerson(Comp comp) {
        this.comp = comp;
    }

    @Override
    public int compare(Person p1, Person p2) {

        int rYear = p1.getYear() - p2.getYear();
        int rName = p1.getName().compareTo(p2.getName());

        int result = 0;

        switch (comp) {
            case BY_NAME:
                result = rName != 0 ?  rName : rYear; break;
            case BY_NAMERev:
                result = rName != 0 ? -rName : rYear; break;
            case BY_YEAR:
                result = rYear != 0 ?  rYear : rName; break;
            case BY_YEARRev:
                result = rYear != 0 ? -rYear : rName; break;
        }
        return result;
    }
}
```

Then we can use objects of this class as a comparator to sort collections (or arrays) of **Persons** in various ways:

Listing 21 ELM-Comps1/Main.java

```
import java.util.Arrays;

public class Main {

    public static void main(String[] args) {
        Person[] persons = {
                               new Person("Mary",1990),
                               new Person("Joan",1992),
                               new Person("Suzy",1992),
                               new Person("Beth",1992),
                               new Person("Suzy",1980),
                               new Person("Katy",1982),
                           };
        Person.show(persons,"At the beginning:");

        Arrays.sort(persons);
        Person.show(persons,"Natural order: " +
                      "by year, then by name");

        Arrays.sort(persons,
            new CompPerson(CompPerson.Comp.BY_NAME));
        Person.show(persons,"Order BY_NAME: " +
                      "by name, then by year");

        Arrays.sort(persons,
            new CompPerson(CompPerson.Comp.BY_NAMERev));
        Person.show(persons,"Order BY_NAMERev: " +
                "by name reversed, then by year");

        Arrays.sort(persons,
            new CompPerson(CompPerson.Comp.BY_YEAR));
        Person.show(persons,"Order BY_YEAR: " +
                      "by year, then by name");

        Arrays.sort(persons,
            new CompPerson(CompPerson.Comp.BY_YEARRev));
        Person.show(persons,"Order BY_YEARRev: " +
                "by year reversed, then by name");

        Arrays.sort(persons,
                   (f,s) -> s.getYear() - f.getYear());
        Person.show(persons,"Order by lambda : " +
                "by year ");
    }
}
```

The program prints

```
At the beginning:
Mary(1990) Joan(1992) Suzy(1992) Beth(1992) Suzy(1980) Katy(1982)

Natural order: by year, then by name
Suzy(1980) Katy(1982) Mary(1990) Beth(1992) Joan(1992) Suzy(1992)

Order BY_NAME: by name, then by year
Beth(1992) Joan(1992) Katy(1982) Mary(1990) Suzy(1980) Suzy(1992)

Order BY_NAMERev: by name reversed, then by year
Suzy(1980) Suzy(1992) Mary(1990) Katy(1982) Joan(1992) Beth(1992)

Order BY_YEAR: by year, then by name
Suzy(1980) Katy(1982) Mary(1990) Beth(1992) Joan(1992) Suzy(1992)
```

```
Order BY_YEARRev: by year reversed, then by name
Beth(1992) Joan(1992) Suzy(1992) Mary(1990) Katy(1982) Suzy(1980)

Order by lambda : by year
Beth(1992) Joan(1992) Suzy(1992) Mary(1990) Katy(1982) Suzy(1980)
```

1.3 Inner and anonymous classes

It is possible to define a class inside another class — we then say that it is an **inner class**; the class in which an inner class is defined is its **outer** (or **surrounding**) class. An inner class may be defined as static or non-static.

1.3.1 Non-static inner classes

Let us consider *non-static* inner classes first. Objects of an inner class cannot be created independently of objects of its surrounding class: they always contain a reference to a "parent" object of the outer class — this reference is accessible under the name Outer.this, where **Outer** is the name of the surrounding class. Therefore, such objects may only be created inside methods of the outer class (and **this** will be equivalent to Outer.this inside the object created) or by invoking **new Inner(...)** on an object of the outer class.

What is also important is the fact that both classes, an inner class and its surrounding class, are "friends", i.e., all members, even private, of one of them are directly accessible by methods of the other, what is illustrated in the example below:

Listing 22 ELH-OutInn/OutInn.java

```java
class Outer {
    private String sOut;
    Outer(String s) { sOut = s; }

    class Inner {
        private String sInn;
        Inner(String s) { sInn = s; }
        @Override
        public String toString() {
            return "Inner-" + sInn + " parent " +
                   "Outer-" + Outer.this.sOut; // <- syntax!
                                               // note that
                                               // sOut is
                                               // private!
        }
    }

    public Inner getInner(String i) {
        Inner inn = new Inner(i);
        System.out.println("Creating inner " + inn.sInn);
        return inn;
    }

    @Override
    public String toString() { return "Outer-" + sOut; }
}

public class OutInn {
    public static void main (String[] args) {
        Outer out1 = new Outer("out1");
        Outer.Inner inn1 = out1.getInner("inn1");
        Outer.Inner inn2 = out1.new Inner("inn2");
        System.out.println(out1);
        System.out.println(inn1);
        System.out.println(inn2);
        System.out.println(out1.getClass().getName());
        System.out.println(inn1.getClass().getName());
        System.out.println(inn2.getClass().getName());
    }
}
```

which prints

```
Creating inner inn1
Outer-out1
Inner-inn1 parent Outer-out1
Inner-inn2 parent Outer-out1
Outer
Outer$Inner
Outer$Inner
```

1.3.2 Static inner classes

An inner class may also be declared as static. It is still a “friend” of the outer class, but there is no Outer.this inside the object of the inner class; therefore, objects of the inner class may exist independently of any objects of the outer class.

In the example below class MyStack represents a stack (of integers) implemented as a singly linked list. We need a class representing individual nodes of the list, but the user of the stack doesn’t need to know about its existence; therefore we define Node as a private static inner class inside MyStack:

Listing 23 GMC-StackSimple/MyStack.java

```
public class MyStack {
      // static inner class
    private static class Node {
        int  data;
        Node next;
        Node(int d, Node n) {
            data = d;
            next = n;
        }
    }
```

```java

    private Node top;

    public MyStack() {
        top = null;
    }
    public void push(int d) {
        top = new Node(d, top);
    }
    public int pop() {
        int d = top.data;
        top = top.next;
        return d;
    }
    public boolean empty() {
        return top == null;
    }
}
```

and the user uses only objects of type **MyStack**; class **Node** is just an “implementation detail”:

Listing 24 GMC-StackSimple/StackSimple.java

```java
public class StackSimple {
    public static void main (String[] args) {
        MyStack stInt = new MyStack();
        for (int i = 5; i > 0; --i)
            stInt.push(i);
        while (!stInt.empty())
            System.out.print(stInt.pop() + " ");
        System.out.println();
    }
}
```

1.3.3 Anonymous classes

Sometimes we have to create just one object of a type which implements an interface, or extends an abstract class, or behaves as an object of an existing concrete class but with one or a few methods overridden: in such situation one can use object of an **anonymous class**. The syntax is illustrated in the example below: after **new** we specify a class (concrete or abstract) that we want our anonymous class to extend, or an interface that we want it to implement. In the first case we can also pass arguments to a constructor; in any case round parentheses are obligatory. Then, in curly braces, we write an implementation (normally, we just override one or more methods). The compiler will then create an anonymous class and return an object of this type:

Listing 25 ELI-Anon/Anon.java

```java
interface BiIntOperator {
    int apply(int i, int j);
}

class AddAndMult implements BiIntOperator {
    int seed;
    AddAndMult(int seed) { this.seed = seed; }
    AddAndMult()         { this(1);          }
    @Override
    public int apply(int i, int j) { return seed*(i + j); }
}

class MultAndAdd implements BiIntOperator {
    int seed;
    MultAndAdd(int seed) { this.seed = seed; }
    MultAndAdd()         { this(1);          }
    @Override
    public int apply(int i, int j) { return seed + i*j; }
}

public class Anon {
    public static void main(String[] args) {
        BiIntOperator[] opers = {
              // objects of concrete classes
              // implementing an interface
            new AddAndMult(2),
            new MultAndAdd(5),
              // object of anonymous class
              // implementing an interface
            new BiIntOperator() {
                @Override
                public int apply(int i, int j) {
                    return i*i + j*j;
                }
            },
              // object of anonymous class
              // extending a 'normal' class
            new MultAndAdd(3) {
                @Override
                public int apply(int i, int j) {
                    return seed*(i*i + j*j);
                }
            }
        };
        int a = 1, b = 2;
        for (BiIntOperator op : opers)
            System.out.print(op.apply(a,b) + " ");
        System.out.println();
    }
}
```

(the program prints 6 7 5 15).
Of course, we cannot create another object of such an anonymous class, because it doesn't even have a name. For the same reason, anonymous classes cannot define any constructors.

Another example illustrating abstract and anonymous classes: we define an abstract class **Animal**

Listing 26 EQE-AbstractAnimals/Animal.java

```
public abstract class Animal {
    String name;
    double weight;

    public Animal(String name, double weight) {
        this.name   = name;
        this.weight = weight;
    }

    abstract public String speak();

    @Override
    public String toString() {
        return name + "(" + weight + ") - " + speak();
    }
}
```

and then we use it in a program creating various animals

Listing 27 EQE-AbstractAnimals/Main.java

```
public class Main {
    public static void main(String[] args) {

        Animal max =
            new Animal("Max", 15) {
                @Override
                public String speak() {
                    return "bow-wow";
                }
            };

        Animal[] animals =
            {
                max,
                new Animal("Batty", 3.5) {
                    @Override
                    public String speak() {
                        return "miaou-miaou";
                    }
                }
            };

         for (Animal a : animals)
             System.out.println(a);
    }
}
```

Important: When we define an anonymous class, local variables visible in the current scope will be accessible (and can be used) inside the body of methods of the anonymous class being created. The only condition is that these local variables are

- declared as **final** (and, of course, initialized), or
- **effectively final**, i.e., they are defined and initialized and then not modified.

1.4 Lambdas

Instead of passing an object of a concrete or anonymous class implementing an interface, one can often provide just a simple expression describing a functionality of a method which is supposed to be implemented (we call such an expression a **lambda**). For this to be possible, the compiler has to know which method it is and from which interface. Therefore, we can use a lambda only when it is clear from the context implementation of which interface is expected. In order for the compiler to know what method is to be overridden, there must be only one abstract method in the interface involved — such interfaces, with only one abstract method, are called **functional interfaces**.

A lambda expression itself is composed of three parts:

- List of parameters, as in declaration of a 'normal' function. However, usually the types of parameters need not be specified, because the compiler is able to deduce them from the context. If there is only one parameter and type is omitted, parentheses are optional. If the list of parameters is empty, we just write empty parentheses.
- The "arrow" token: $- >$.
- A function body enclosed in curly braces. If the body contains just one expression (i.e., something that has a value), there are no braces, no semicolon at the and no **return** statement: the value of the expression will be evaluated and returned — return type will be deduced as the type of this value. Braces may be also omitted if the body consists of one statement which does not have any value — return type **void** will then be assumed.

Examples:

```
(int x, int y) -> x + y
(x, y) -> x*y < 0
() -> Math.random()
e -> System.out.println(e)
```

In these examples we assume that the compiler will be able to infer all necessary types and deduce what functional interface is to be implemented.

The example below illustrates both cases: when the body of a lambda is a single expression (no semicolon, no return, no braces) and when it is implemented as a compound statement (with braces, return and semicolons after statements, as usually). Note that the compiler expects, as the second argument of the sort function 'something that implements the Comparator interface'. As what is to be sorted is an array of Persons, the compiler knows also that it should be in fact Comparator<Person>, so the type of p1 and p2 is deduced to be Person:

Listing 28 EMD-InterF/InterF.java

```java
import java.util.Arrays;

class Person {

    private String name;
    private int    year;

    public Person(String name, int year) {
        this.name = name;
        this.year = year;
    }

    public String getName() { return name; }
    public int getYear()    { return year; }

    @Override
    public String toString() {
        return name + "(" + year + ")";
    }

    static void show(Person[] persons, String message) {
        System.out.println(message);
        for (Person person : persons)
            System.out.print(person + " ");
        System.out.println();
    }
}

public class InterF {

    public static void main(String[] args) {
        Person[] persons =
            { new Person("Mary",1990),
              new Person("Joan",1992),
              new Person("Suzy",1992),
              new Person("Beth",1992),
              new Person("Suzy",1980),
              new Person("Katy",1982), };
        Person.show(persons,"At the beginning:");

```

```
          // lambda as a single expression -
          // no return, no semicolon
        Arrays.sort(persons,
            (p1, p2) -> p1.getYear()-p2.getYear());
        Person.show(persons, "Ordered by age");

          // lambda as a compound statement -
          // return and semicolons, as usually
        Arrays.sort(persons, (p1, p2) ->
            {
                int d = p1.getName().compareTo(p2.getName());
                if (d != 0) return d;
                return p1.getYear() - p2.getYear();
            });
        Person.show(persons,"Ordered by name then age");
    }
}
```

The program prints

```
At the beginning:
Mary(1990) Joan(1992) Suzy(1992) Beth(1992) Suzy(1980) Katy(1982)
Ordered by age
Suzy(1980) Katy(1982) Mary(1990) Joan(1992) Suzy(1992) Beth(1992)
Ordered by name then age
Beth(1992) Joan(1992) Katy(1982) Mary(1990) Suzy(1980) Suzy(1992)
```

The scoping rules for lambdas are somewhat special. We can imagine that the compiler creates an object of an anonymous class implementing an interface and then implements its abstract method based on our lambda expression. However, it is not so — the body of a lambda behaves as a block inside the function it is defined in. This fact has several consequences, among others these

- inside the body of a lambda, one cannot define variables with same name as local variables from the surrounding scope;
- the reference **this** used inside the body of a lambda refers to an object of the surrounding class, *not* to an object of an anonymous class.

1.5 More examples

Let's look at some examples.

Listing 29 ELU-FInter/FInter.java

```
@FunctionalInterface
interface Calc {
    boolean test(Double d);
}

@FunctionalInterface
```

```java
interface Cons {
    void consume(Object ob);
}

public class FInter {
    public static void main (String[] args) {
        double mn = 0, mx = 10; // effectively final
        Calc[] arr = {
            d -> d <  0,         // type of d inferred
            d -> d >= 0,
            d -> mn <= d && d <= mx
        };

        Cons cons = ob -> System.out.print(ob + " ");

        for (double d = -2; d < 15; d += 5) {
            for (Calc calc : arr)
                cons.consume(calc.test(d));
            System.out.println();
        }
    }
}
```

And another simple example. Here, we define an interface which declares a method **transform** which 'transforms' a strings — it takes a string and returns another string. Note that the function **transArray** in class **SimpleInter** takes, as its second argument 'something that implements **Transformation** interface'. In **main** we call this function passing this 'something' in three different ways:

- as an object of a separate class **Reverse** implementing **Transformation**;
- as an object of an anonymous class implementing the same interface;
- as a lambda.

Listing 30 ELK-SimpleInter/SimpleInter.java

```java
import java.util.Arrays;

interface Transformation {
    String transform(String arg);
}

class Reverse implements Transformation {
    @Override
    public String transform(String s) {
        char[] a = s.toCharArray();
        for (int i = 0, j = s.length()-1; i < j; ++i, --j) {
            char c = a[i];
            a[i] = a[j];
            a[j] = c;
        }
        return new String(a);
    }
}

public class SimpleInter {
    private static void transArray(String[] array,
                                   Transformation t) {
        for (int i = 0; i < array.length; ++i)
            array[i] = t.transform(array[i]);
    }

    public static void main (String[] args) {
        String[] arr = {"Mary", "Alice", "Janet", "Rachel"};
        System.out.println(Arrays.toString(arr));

          //object of a class
        transArray(arr, new Reverse());
        System.out.println(Arrays.toString(arr));

          // object of an anonymous class
        transArray(arr,
                   new Transformation() {
                       @Override
                       public String transform(String s) {
                           return s.toUpperCase();
                       }
                   });
        System.out.println(Arrays.toString(arr));

          // lambda
        transArray(arr, s -> "" + s.charAt(0));
        System.out.println(Arrays.toString(arr));
    }
}
```

The program prints

```
[Mary, Alice, Janet, Rachel]
[yraM, ecilA, tenaJ, lehcaR]
[YRAM, ECILA, TENAJ, LEHCAR]
[Y, E, T, L]
```

Another important example of using a lambda has been already shown in one of the previous examples — see Listing 21. The **sort** function called with two arguments expects as the second one ‘something implementing the **Comparator** interface’. This interface *is* a functional interface, because it declares only one abstract method: **compare**. Therefore, as the implementation of this method is usually rather short, it is very convenient to use lambdas instead of creating separate classes whose only purpose is to ’wrap’ the **compare** method.

In the examples below, we use the called *generic* classes and functions; their full understanding will require the knowledge from chapter 2, p. 33).

The first example:

Listing 31 ELW-LambdaInter/MyBiOp.java

```java
import java.util.ArrayList;
import java.util.List;

@FunctionalInterface
interface MyBiOpInterface<T> {
    T apply(T a, T b);
}

class Mult implements MyBiOpInterface<Double> {
    @Override
    public Double apply(Double a, Double b) { return a*b; }
}

public class MyBiOp {
    public static void main (String[] args) {
        List<MyBiOpInterface<Double>>
                    opers = new ArrayList<>();

          // addition - reference to (static) method
        opers.add( Double::sum );

          // subtraction - anonymous class
        opers.add( new MyBiOpInterface<Double>() {
            @Override
            public Double apply(Double a, Double b) {
                return a - b;
            }
        });

          // multiplication - object of a named class
          // implementing the MyBiOpInterface interface
        opers.add( new Mult() );

          // division - lambda
        opers.add( (a,b) -> a/b );

          // with closure ('shift' is effectively final)
        int shift = 10;
        opers.add( (a,b) -> a/b + shift);

        for (MyBiOpInterface<Double> op : opers)
            System.out.println(op.apply(10.5,3.5));
    }
}
```

and another one

Listing 32 ELX-FuncInter/FuncInter.java

```java
@FunctionalInterface
interface MyInterface<T> {
    int len(T t);
}

public class FuncInter {
    static <T> int calc(T arg, MyInterface<T> f) {
        return f.len(arg);
    }

    public static void main (String[] args) {
        String s = "Alice";
        int result1 = calc(
            s,
            new MyInterface<String>() {
                @Override
                public int len(String s) {
                    return s.length();
                }
            });
        System.out.println("result1 = " + result1);

        Double d = 123.456; // boxing
        int result2 =
            calc(d, v -> v.toString().length());
        System.out.println("result2 = " + result2);

        int result3 =
            calc(d, v -> (int)(1000*v+4));
        System.out.println("result3 = " + result3);
    }
}
```

The following example is very similar to that in Listing 38, but now we use lambdas: directly as an argument, or to create a variable which could be used more than once:

Listing 33 EMA-IFacLam/IFacesLam.java

```java
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

interface Operat<T> {
    T oper(T lhs, T rhs);
}

```

```
class Tools {
    public static <T> T foldl(
            Operat<T> op, List<T> list, T id) {
        T acc = id;
        for (T e : list)
            acc = op.oper(acc,e);
        return acc;
    }

    public static <T> List<T> combine(
            Operat<T> op, List<T> l1, List<T> l2) {
        List<T> res = new ArrayList<T>();
        int size = Math.min(l1.size(), l2.size());
        for (int i = 0; i < size; ++i)
            res.add(op.oper(l1.get(i), l2.get(i)));
        return res;
    }
}

public class IFacesLam {
    public static void main (String[] args) {
        Operat<String> operStr = (lhs, rhs) -> lhs + rhs;

        List<Integer> listInt1  = Arrays.asList(1,2,3,4),
                      listInt2  = Arrays.asList(5,6,7,8);
        List<Integer> intRes =
            Tools.combine((lhs, rhs) -> lhs + rhs,
                          listInt1, listInt2);
        System.out.println("intRes = " + intRes);

        List<String> listStr1  = Arrays.asList("a","b","c"),
                     listStr2  = Arrays.asList("1","2","3");
        List<String> strRes =
            Tools.combine(operStr, listStr1, listStr2);
        System.out.println("strRes = " + strRes);

        int intFold = Tools.foldl(
                (lhs, rhs) -> lhs + rhs, intRes, 0);
        System.out.println("intFold = " + intFold);

        String strFold = Tools.foldl(operStr, strRes, "");
        System.out.println("strFold = " + strFold);
    }
}
```

The last example illustrates an interface with one *defined* default method; note, that we have to inform the compiler that S is an additional type parameter (T and R have already been declared as such.) The interface represents an operation (mapping), called here **apply**, from values of one type to values of another type (T→R). However, the interface *defines* also a method (**compos**). This is a method, so it is called on an object representing one operation (say, f : **T**→**R**), takes a value arg of type **T** as an argument and an object representing another operation (say, g : **R**→**S**). It then returns the result of the composition $(g \circ f)(\mathrm{arg})$. The following program

Listing 34 EMB-Compos/Composit.java

```java
@FunctionalInterface
interface Func<T,R> {
    R apply(T e);
    default <S> S compos(T arg, Func<R,S> g) {
        return g.apply(apply(arg));
    }
}

public class Composit {
    public static void main(String[] args) {
        Func<String,Integer> f = s -> s.length();
        System.out.println("g(f(\"abc\")) = " +
                f.compos("abc", v -> v*Math.PI));
    }
}
```

prints

```
        g(f("abc")) = 9.42477796076938
```

as the length of "abc" is 3 and $3\pi \approx 9.42477796076938$. Note that the g operation was given as a lambda but still the compiler was able to deduce that the type **S** of the result should be **Double**.

Section 2

Introduction to generic classes

This section covers a few basic concepts related to generics, or parametrized classes, in Java. The subject is not easy; generics were added to Java rather late and their implementation is quite complicated, as it had to be consistent with earlier versions of the language. Let us also mention here that parametrized classes are somewhat related to class templates in C++ and Ada, but this similarity is rather misleading and superficial; it definitely should not be taken too literally.

2.1 Type parameters

Parametrized class uses as names of types arbitrary identifiers (type parameters) which are then concretized, when we use objects of the generic type. Then we have to tell the compiler what concrete class this identifier should denote. We introduce type parameters of a class like this:

```
class Pair<F,S> {
    // here we use F and S as names of types
}
```

Here F and S stand for names of some, as yet unknown, types. In the definition of the class we can use names F and S as names of classes (but *not* primitive types!). Of course, there are no classes F or S, so when creating an object of our class, we have to specify (in angle brackets) what these names should stand for:

```
Pair<Integer,String> p =
           new Pair<Integer,String>(2,"Sue");
  // or just (using diamond operator)
Pair<Integer,String> q = new Pair<>(1,"Lea");
```

In the second case, we used “type inferring” (diamond operator); we don’t need to repeat types on the right-hand side, because the compiler already knows them looking at the left-hand side (but angle brackets, although empty, are still required).

However, in the code produced by the compiler, type parameter will *not* be present: at run time types denoted by, say, T will be just **Object**. Information about the type of T is only used at compile time.

Let us consider, as an example, the following program:

Listing 35 GMA-GenerPair/Pair.java

```
import java.lang.reflect.Method;

public class Pair<F, S> {
    private static int count = 0;
    private F first;
    private S second;
    public Pair(F f, S s) {
        count++;
        first  = f;
        second = s;
    }
    public F getFirst()  {return first; }
    public S getSecond() {return second;}
    public void setFirst(F f)  {first  = f;}
    public void setSecond(S s) {second = s;}
    public String toString() {return first + " " + second;}

    public static void main(String[] args) throws Exception {
        Pair<String, Integer> p1 = new Pair<>("A",1);
        Pair<String, String>  p2 = new Pair<>("C", "D");

          // what are the dynamic types of p1 and p2 ?
        System.out.println("Class of p1: " + p1.getClass() +
                "; Class od p2: " + p2.getClass());

          // method signatures
        for (Method m : p1.getClass().getDeclaredMethods())
            if (!m.getName().equals("main"))
                System.out.println(m); // type erasure

          // only one static member 'count'
        System.out.println(p1.count + " " + p2.count);

          // casting not needed, autoboxing int -> Integer
        p1.setSecond(2);

          // automatic unboxing Integer -> int
        int i = p1.getSecond();

          // no casting, must be a String
        String s = p2.getFirst();

        System.out.println("p1.second = " + i);
        System.out.println("p2.first  = " + s);
    }
}
```

As we can see, the compiler remembers types used when creating objects, so casting is not needed; calling **getFirst** on p2 must give us a **String** and invoking **setSecond** on p1 we pass an **int** and the compiler knows that it has to be converted to **Integer**. However, the dynamic type, or, strictly speaking, the so called “raw” type of both p1 and p2 is just **Pair**. We can see it from the output:

```
Class of p1: class Pair; Class od p2: class Pair
public java.lang.Object Pair.getSecond()
public void Pair.setFirst(java.lang.Object)
public void Pair.setSecond(java.lang.Object)
public java.lang.String Pair.toString()
public java.lang.Object Pair.getFirst()
2 2
```

```
p1.second = 2
p2.first  = C
```

Therefore, at runtime, there exist only one type, namely just Pair of Objects. Notice that at runtime arguments of setters (setFirst and setSecond) and return type of getters (getFirst and getSecond) are just Object: this is known as **type erasure**. Only at compile time the compiler knows the required (declared) types of the two elements of a pair and will not allow us to insert (for example using setFirst) an object of a wrong type as this pair's first or second component. Also notice that there exist only one static member `count` because dynamic types of both `p1` and `p2` are the same — just Pair.

Parametrized type can be, and are, useful, because they

- allow us to avoid explicit conversions, because the compiler knows expected types of arguments and return values of methods;
- make the code shorter and simpler to read and understand;
- allow the compiler to detect many errors related to type mismatch, which would otherwise manifest themselves at runtime, trigering, for example, ClassTypeExceptions.

On the other hand, mainly because of the type erasure that we have just mentioned, there are some quite severe restrictions when defining and using parametrized types. If T is a type parameter

- We cannot create objects of type T;
- We cannot create arrays of (references to) objects of type T (but it *is* possible to create collections parametized by T);
- We cannot use instanceof operator with T;
- Type T cannot be used for static fields (as parametrized class corresponds to one raw type);

Let us look at another example of a parametrized class: a class representing a queue. Making our class generic (parametrized) allows us to create queues of elements of various types preserving strict type checking:

Listing 36 GME-QueueGener/MyQueue.java

```java
public class MyQueue<E> {

    private class Node {
        E    data;
        Node next = null;
        Node(E d) { data = d; }
    }
    private Node head, tail;

    public MyQueue() {
        head = tail = null;
    }
    public void enqueue(E d) {
        if (head == null)
            head = tail = new Node(d);
        else
            tail = tail.next = new Node(d);
    }
    public E dequeue() {
        E d = head.data;
        if ((head = head.next) == null) tail = null;
        return d;
    }
    public boolean empty() {
        return head == null;
    }
}
```

In **main** we create queues of **String**s and **Double**s and use them in a uniform way:

Listing 37 GME-QueueGener/QueueGener.java

```
public class QueueGener {
    public static void main (String[] args) {
        MyQueue<String> queueS = new MyQueue<>();
        MyQueue<Double> queueD = new MyQueue<>();
        for (double d = 0.5; d < 5; d += 1) {
            queueS.enqueue(String.valueOf(d));
            queueD.enqueue(d); // boxing
        }
        while (!queueS.empty() && !queueD.empty()) {
              // no casting required
            String s = queueS.dequeue();
            double d = queueD.dequeue();
            System.out.println(
                    "String: " + s +  "   " +
                    "Double: " + d);
        }
    }
}
```

In the program, class **MyQueue** implements a queue of objects of any type. This does not mean that we can enqueue any object to any queue: when creating a queue, we decide what the type of enqueued objects should be. For example, the queue queueS is declared as queue of **Strings**. Therefore, the compiler will not allow us to enqueue anything other than **String** (and in the case of queueD — **Doubles**). Also notice that when enqueueing and dequeueing elements, we don't have to use casting: the compiler knows what type is expected and will even automatically perform boxing/unboxing of primitive types for us.

Let us consider another example. We define a generic interface **Operat** which declares one function representing an operator, i.e. a function taking two arguments of the same type and returning a result of the same type. Object implementing this interface will then be used by static functions defined in class **Tools**. Note, how we define generic *static functions* — here we inform the compiler that **T** is not a class but a type parameter for every static function defined in the class separately; one of these functions performs the so called *left-folding* and the other combines two sequences into one. We don't create any concrete classes implementing **Operat**; instead, in **main**, we just pass objects of anonymous classes implementing it:

Listing 38 ELZ-IFac/IFaces.java

```java
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

interface Operat<T> {
    T oper(T lhs, T rhs);
}

class Tools {
    public static <T> T foldl(
            Operat<T> op, List<T> list, T id) {
        T acc = id;
        for (T e : list)
            acc = op.oper(acc,e);
        return acc;
    }

    public static <T> List<T> combine(
            Operat<T> op, List<T> l1, List<T> l2) {
        List<T> res = new ArrayList<T>();
        int size = Math.min(l1.size(), l2.size());
        for (int i = 0; i < size; ++i)
            res.add(op.oper(l1.get(i), l2.get(i)));
        return res;
    }
}

public class IFaces {
    public static void main (String[] args) {
        Operat<Integer> operInt = new Operat<Integer>() {
            @Override
            public Integer oper(Integer lhs, Integer rhs) {
                return lhs + rhs;
            }
        };
        Operat<String> operStr = new Operat<String>() {
            @Override
            public String oper(String lhs, String rhs) {
                return lhs + rhs;
            }
        };

        List<Integer> listInt1  = Arrays.asList(1,2,3,4),
                      listInt2  = Arrays.asList(5,6,7,8);
        List<Integer> intRes =
            Tools.combine(operInt, listInt1, listInt2);
        System.out.println("intRes = " + intRes);

        List<String> listStr1  = Arrays.asList("a","b","c"),
                     listStr2  = Arrays.asList("1","2","3");
        List<String> strRes =
            Tools.combine(operStr, listStr1, listStr2);
        System.out.println("strRes = " + strRes);

        int intFold = Tools.foldl(operInt, intRes, 0);
        System.out.println("intFold = " + intFold);

        String strFold = Tools.foldl(operStr, strRes, "");
        System.out.println("strFold = " + strFold);
    }
}
```

The program prints

```
intRes = [6, 8, 10, 12]
strRes = [a1, b2, c3]
intFold = 36
strFold = a1b2c3
```

2.2 Bounded types

Specifying a type parameter, say T, we can limit possible types that it can represent to types meeting some requirements. This allows us, for example, to use, on objects of T, methods not necessarily only from class Object but more specific methods from other classes that has been specified as type bounds. Such restrictions define sets of classes that can be substituted for T: then we can use methods which the classes from these sets contain. The syntax looks like this:

```
    T extends Type1 & Type2 & Type3  & ... & TypeN
```

where:

- T is the type parameter;
- Type1 is the name of a class or an interface;
- Type2 ... TypeN are names of interfaces.

As we can see, if among restrictions there is a (concrete or abstract) class, it must be specified as the first in the list, all others bounds must be interfaces. Types Type1 ... TypeN may be themselves parametrized (generic) classes or interfaces.

Objects of types restricted in this way can be used as objects of type Type1: compilation will fail if we try to substitute for T something that does not extend or implement Type1 or does not implement Type2 ... TypeN. Therefore, it will be safe to call methods of all these restricting classes or interfaces.

In the example below, class **GenArr** is parametrized by type **T**, but not arbitrary but only by classes implementing **Comparable** — therefore, we can use the method **compareTo** on objects from the array (in order to find the minimum and maximum elements of the array):

Listing 39 GMH-MinMaxGener/MinMaxGener.java

```java
class GenArr<T extends Comparable<T>> {
    private T[] arr;
    private T   min, max;

    public GenArr(T[] arr) {
        if (arr == null || arr.length == 0)
            throw new IllegalArgumentException();
        min = arr[0];
        max = arr[0];
        for (int i = 1; i < arr.length; ++i) {
              // we can use compareTo, as we know
              // that T extends Comparable
              // (and the compiler has checked it)
            if (arr[i].compareTo(min) < 0) min = arr[i];
            if (arr[i].compareTo(max) > 0) max = arr[i];
        }
    }
    public T getMin() { return min; }
    public T getMax() { return max; }
}

public class MinMaxGener {
    public static void main(String[] args) {
        GenArr<Integer> mnmxI =
            new GenArr<>(new Integer[]{3, -2 , -7, 2});
        GenArr<String> mnmxS =
            new GenArr<>(new  String[]{"A", "Z", "C"});

        System.out.println("I - min = " + mnmxI.getMin() +
                           "\nI - max = " + mnmxI.getMax());
        System.out.println("S - min = " + mnmxS.getMin() +
                           "\nS - max = " + mnmxS.getMax());
    }
}
```

It is also possible to parametrize just a method (also static), not the whole class. We then specify the name of the type parameter (in angle brackets) just before the return type of the method in question:

```
access_specifier [static] <T> return_type name(parameter_list)
```

For example, in the following program, the class **MinMaxMeth** is *not* generic, however the method **getMax** is: it may take an array of elements of any type, as long as this type implements **Comparable**:

Listing 40 GMI-MinMaxMeth/MinMaxMeth.java

```java
class MinMaxMeth {
    static <T extends Comparable<T>> T getMax(T[] arr) {
        if (arr == null || arr.length == 0)
            throw new IllegalArgumentException();
        T max = arr[0];
        for (int i = 1; i < arr.length; ++i)
            if (arr[i].compareTo(max) > 0) max = arr[i];
        return max;
    }

    public static void main(String[] args) {
        int mxi = getMax(new Integer[]{3, -2 , -7, 2});
          // one may enforce type to be substituted for T;
          // usually, as here, not needed, as the correct
          // type will be inferred by the compiler anyway
        String mxs = MinMaxMeth.<String>getMax(
                new  String[]{"A", "Z", "C"});

        System.out.println("I - max = " + mxi);
        System.out.println("S - max = " + mxs);
    }
}
```

Section 3

Enum types

Enum types roughly correspond to enumerations in C++, but are implemented in a different way: they are all *objects*, in contrast to C++ where they are always backed by an integer type. They are very useful in situations when we need a type with only very limited number of possible values.

3.1 Basic definitions

Defining an enum we define a new type (just like defining a class). However, this type is somewhat special: limited number of *unmodifiable* enum constants (which are implements as immutable objects) of this type is created when an enum class is loaded by the JVM and then it is impossible to create any more such objects. Therefore, we can say, that this type defines just a — usually small — set of constants. In fact, we know another example of type with only a few possible values: type **boolean** has only two — **true** and **false**. However, **boolean** is a primitive type, while enum constants are full-fledged objects.

Each of these constants has a fixed name, as is the case for **boolean**s, and can be in fact a singleton of a *different* class.

Enums can be useful for defining types that by their very nature have only a small number of possible values: there are only two sexes, four seasons, seven Wonders of the World, four card suits and four Horsemen of the Apocalypse. Without enums, we could just assign numbers to them; for example, in a class or an interface we could write

```
final static int CLUBS    = 0,
                 DIAMONDS = 1,
                 HEARTS   = 2,
                 SPADES   = 3;
```

but this poses a lot of problems

- if a function takes a card suit as its parameter, it has to declare it as an **int**. But then nothing can prevent us from passing, for example, 129 as the argument, which wouldn't make any sense;
- of course, all functions operating on card suits could check if the value passed to them is in the range $[0, 3]$, but then it would be hard to add a new suit (joker. . .) — we would have to correct the checking and interpretation in many places;
- in many situations we would have to remember the interpretation of the numbers; for example, printing just numbers as they are wouldn't be very informative — we would have to "convert" them manually into **String**s or do something equivalent.

Let us look at an example:

Listing 41 CYG-Enum1/EnumEx1.java

```
public class EnumEx1 {

    public enum Season {SPRING, SUMMER, FALL, WINTER};
```

```java
    public static void main (String[] args) {
        Season[] seasons = Season.values();            //+1
        for (Season s : seasons)
            System.out.print(s + " ");                 //+2
        System.out.println();

        System.out.println(Season.WINTER + " is the " +
            (Season.WINTER.ordinal()+1) + "th season");//+3
        Season f = Season.valueOf("FALL");             //+4
        System.out.println("FALL is " + f + "...");
        System.out.println("Is f equal to FALL? " +
                              (f == Season.FALL));     //+5
        for (Season s : seasons)
                System.out.print(german(s) + " ");
        System.out.println();
    }

    private static String german(Season s) {
        return switch (s) {                            //+6
            case SPRING -> "Fr\u00fchling";
            case SUMMER -> "Sommer";
            case FALL   -> "Herbst";
            case WINTER -> "Winter";
        };
    }
}
```

The program prints

```
SPRING SUMMER FALL WINTER
WINTER is the 4th season
FALL is FALL...
Is f equal to FALL? true
Frühling Sommer Herbst Winter
```

We define an enum type **Season**. The definition is inside a class here, but this is not important; equally well we could have defined the enum (with **public** or default accessibility) in a separate file. This enum has exactly four values corresponding to four, and only four, objects: SPRING, SUMMER, FALL, WINTER. What can we do with type **Season** and its constants?

- static function **values()** returns an array of all (four in our case) constants of the enum (line //+1);
- method **toString()** is automatically overridden and returns the name of the enum constant (it is used in line //+2);
- we can call **ordinal()** on enum constants (remember that these are objects) and get their 'index' — starting from 0 and in the order as they were defined (//+3);
- static function **valueOf(String name)**) returns enum constant named name; if there were no constant with this name, exception would have been thrown (//+4).

If we have two references to the same enum constant, they are exactly equal, i.e., they point to the same object, because each enum constant is represented by exactly one object. Therefore, to compare enum constant we don't need (although we can) to use method **equals**: just '==' or '!=' are safe and sufficient (//+5).

- as integer types and Strings, enum constants may be used in switch statements (//+6). In case clauses we don't have to use full names (like Season.SPRING), because the type of the selector (`s` in our case) is known to the compiler.

After compiling the above program

```
java> ls -1 EnumExample1*
    EnumExample1.java
java> javac EnumExample1.java
java> ls -1 EnumExample1*.class
    EnumExample1$1.class
    EnumExample1.class
    EnumExample1$Season.class
```

we can see that the compiler created *one* class for the Season type (which in our case was embedded in the main class, but this is not important here — anyway, a separate class has been created). The program prints

```
SPRING SUMMER FALL WINTER
WINTER is the 4th season
FALL is FALL...
Is f equal to FALL? true
Frühling Sommer Herbst Winter
```

The third line convinces us that indeed two references to FALL are exactly equal, that is they refer to the same object (and therefore can be compared by '==').

Enumerations are also comparable. If `e1` and `e2` are two enumerators of the same enumeration, then they can be compared in the usual way

```
e1.compareTo(e2)
```

The order is determined by the order in which enumerators are declared in the definition of enumeration.

In the following example there are two enumerations describing suits

Listing 42 CYE-SimpEnum/Suit.java

```java
public enum Suit {CLUBS,DIAMONDS,HEARTS,SPADES};
```

and ranks

Listing 43 CYE-SimpEnum/Rank.java

```java
public enum Rank {TWO,THREE,FOUR,FIVE,SIX,SEVEN,
                  EIGHT,NINE,TEN,JACK,QUEEN,KING,ACE};
```

of playing cards. The class Card has two fields, both are enumerators (rank and suit); it also defines one static function:

Listing 44 CYE-SimpEnum/Card.java

```java
public class Card {
    private Rank rank;
    private Suit suit;

    public Card(Rank rank, Suit suit) {
        this.rank = rank;
        this.suit = suit;
    }

    public Rank getRank() { return rank; }
    public Suit getSuit() { return suit; }

    public static Card getHigher(Card c1, Card c2) {
        if      (c1.rank.ordinal() > c2.rank.ordinal())
            return c1;
        else if (c1.rank.ordinal() < c2.rank.ordinal())
            return c2;
        else if (c1.suit.ordinal() > c2.suit.ordinal())
            return c1;
        else
            return c2;
    }

    @Override
    public String toString() {
        return rank + " of " + suit;
    }
}
```

In Main we read data and use our classes

Listing 45 CYE-SimpEnum/Main.java

```java
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        System.out.print("Suits:");
        for (Suit s : Suit.values())
            System.out.print(" " + s );
        System.out.print("\nRanks:");
        for (Rank r : Rank.values())
            System.out.print(" " + r);
        System.out.println();

        Scanner scan = new Scanner(System.in);
        System.out.print("1. rank -> ");
        String r1 = scan.next().toUpperCase();
        System.out.print("1. suit -> ");
        String s1 = scan.next().toUpperCase();
        System.out.print("2. rank -> ");
        String r2 = scan.next().toUpperCase();
        System.out.print("2. suit -> ");
        String s2 = scan.next().toUpperCase();

        Rank rank1 = Rank.valueOf(r1);
        Suit suit1 = Suit.valueOf(s1);
        Rank rank2 = Rank.valueOf(r2);
        Suit suit2 = Suit.valueOf(s2);

        Card card1 = new Card(rank1,suit1);
        Card card2 = new Card(rank2,suit2);

        Card higher = Card.getHigher(card1,card2);
        System.out.println("Card1 = " + card1 +
                         ", Card2 = " + card2 +
                      "\nHigher card: " + higher);
    }
}
```

3.2 Fields, constructors and methods in enumerations

In the previous example, enum constants differ only in their names, so objects representing them were of the same type. However, one can define methods for them, and these methods can have different implementation for each constant separately. First, let us look at the following example:

Listing 46 CYH-EnumMet/EnumEx2.java

```java
public class EnumEx2 {

    enum Season {
        SPRING {                                   //+1
            @Override
            public String toString() {return "Printemps";}
        },
        SUMMER {
            @Override
            public String toString() {
                return "\u00c9t\u00e9";
            }
        },
        FALL {
            @Override
            public String toString() {return "Automne";}
        },
        WINTER {
            @Override
            public String toString() {return "Hiver";}
        }
    };

    public static void main (String[] args) {
        Season[] seasons = Season.values();
        for (Season s : seasons)
                System.out.print(s + " ");          //+2
        System.out.println();
        for (Season s : seasons)
                System.out.print(s.name() + " ");   //+3
        System.out.println();
    }
}
```

Here, *for each constant separately*, in braces just after their names (//+1), we override **toString** method from class **Object**. When we compile this program:

```
java> ls -1 EnumExample2*
    EnumExample2.java
java> javac EnumExample2.java
java> ls -1 EnumExample2*.class
    EnumExample2$1.class
    EnumExample2.class
    EnumExample2$Season$1.class
    EnumExample2$Season$2.class
    EnumExample2$Season$3.class
    EnumExample2$Season$4.class
    EnumExample2$Season.class
```

we can see that the compiler created distinct classes for each constant. This is quite obvious: they cannot be objects of the same class, as it would be impossible to have in one class several implementations of the same method **toString**! Now, when we print s (in line //+2), the overridden version of **toString** will be used. Still, however, the original 'true' name of enum constant may be retrieved by method **name()** (//+3), as we can see from the output

```
Printemps Été Automne Hiver
SPRING SUMMER FALL WINTER
```

Methods that we can define for enum constants are not limited to those inherited. We can define our own methods; moreover, we can add data fields and a constructor (but only one) as well!

In the program below we add two data fields: desc and numOfMonths (line //+3). We also define a standard constructor. Fields and constructors in enumerations are by definition **private**, so we don't even need to declare them as such. Our constructor takes two arguments, so we supply them when defining the enum constants (line //+1):

Listing 47 CYI-EnumAbs/EnumEx3.java

```java
public class EnumEx3 {
    enum Season {
        SPRING("nice",2) {                           //+1
            @Override                                //+2
            public String getDesc() {
                return name() + ": " + desc;
            }
        },
        SUMMER("hot",3) {
            @Override
            public String getDesc() {
                return desc + " in " + name();
            }
        },
        FALL("so, so",4) {
            @Override
            public String getDesc() {
                return name() + "? Well, " + desc;
            }
        },
        WINTER("cold",3) {
            @Override
            public String getDesc() {
                return name() + "! very " + desc;
            }
        }; // <-- semicolon after the last value!

          // fields and constructors are private anyway
        String     desc;                             //+3
        int numOfMonths;
        Season(String d, int i) {                    //+4
            desc        = d;
            numOfMonths = i;
        }

        public int getNumb() { return numOfMonths; } //+5
        public abstract String getDesc();            //+6
    };

    public static void main (String[] args) {
        for (Season s : Season.values())
            System.out.println(s.getNumb() +
                    " months - " + s.getDesc());
    }
}
```

There are also two methods. One of them, **getNumb** has the same common implementation for all objects (line //+5). However, the second one, **getDesc**, is in line //+5 only *declared* (as abstract) and implemented differently for each constant (line //+2). The program prints

```
2 months - SPRING: nice
3 months - hot in SUMMER
4 months - FALL? Well, so, so
3 months - WINTER! very cold
```

Objects representing enum constants are created once only, when the enum is loaded by the JVM, even before initialization of static members of the class, if there are any.

3.3 Enumarations implementing an interface

Enumarator can also implement interfaces. Let us look at an example:

Listing 48 CYK-EnumInterf/CompEnum.java

```java
import java.util.Arrays;
import java.util.Comparator;

public class CompEnum {
    public static void main(String[] args) {
        String[]  arr = {"Alice", "Sue", "Janet", "Bea"};

        Arrays.sort(arr, StrCmp.ByLenAsc);
        System.out.println(Arrays.toString(arr));

        Arrays.sort(arr, StrCmp.ByLenDesc);
        System.out.println(Arrays.toString(arr));

        Arrays.sort(arr, StrCmp.ByLexAsc);
        System.out.println(Arrays.toString(arr));

        Arrays.sort(arr, StrCmp.ByLexDesc);
        System.out.println(Arrays.toString(arr));
    }
}

enum StrCmp implements Comparator<String> {
    ByLenAsc( (s1, s2) -> s1.length() - s2.length()),
    ByLenDesc((s1, s2) -> s2.length() - s1.length()),
    ByLexAsc( (s1, s2) -> s1.compareTo(s2)),
    ByLexDesc((s1, s2) -> s2.compareTo(s1)); // <- semicolon!
    Comparator<String> cmp;                   // field
    StrCmp(Comparator<String> c) {            // constructor
        cmp = c;
    }
    @Override
    public int compare(String s1, String s2) {
        return cmp.compare(s1, s2);
    }
}
```

The enumeration **StrCmp** implements the **Comparator** interface. It has a field cmp of type **Comparator** which will be different for all enumeration constants: it is set in the constructor. When calling the constructor, we have to pass something which *is* a comparator — here, we use different lambdas. Objects of this enumeration can then be used wherever a comparator (of **String**s) is expected, as we can see form the output of the program:

```
[Sue, Bea, Alice, Janet]
[Alice, Janet, Sue, Bea]
[Alice, Bea, Janet, Sue]
[Sue, Janet, Bea, Alice]
```

Section 4

Introduction to collections

Collections represent... well, collections, i.e., aggregates of pieces of data of a given type organized in some form. Objects representing collections in Java have to implement the interface Collection (there is a different kind of collections, map, which implement different interface: Map). All collections are **iterable** (they implement also the Iterable interface) — the importance of this will soon become clear.

We already know interfaces. Generally speaking its something like a class, but with declarations of (abstract) methods — without implementation (although, as we know, it can contain some *default* methods *with* implementation). Other classes may *implement* this interface by providing *definitions* of all the abstract methods. Any class may inherit from (extend) only one class, but can implement many interfaces. Important:

1. Collections (and maps) are always collections of *references* (pointers) — *never* objects themselves!
2. Collections fall into two general categories: **Collection**s and **Map**s.
3. Classes representing collections are generic — they are parametrized by the type of the elements they hold.
4. Their properties and functionality (API) is specified by a hierarchy of interfaces that they implement (i.e., they implement methods declared in these interfaces).

Very often we want a collection of data of a primitive type, like int or double. We cannot insert such data into any collection, because these are not references. However, for each such type, there is a class, objects of which serve as wrappers of such data: Integer for ints, Double for doubles, etc. Normally, we don't even have to create object of these wrapper classes ourselves — this will be done automatically: if a collection is declared as a collection of Integers, we can insert just ints and they will be automatically wrapped into objects of type Integer and put into this collection. Such automatic conversion of values of primitive types to objects is called **boxing**; the reverse operation is called **unboxing**.

When creating objects representing a collection, we should always specify the type of its elements; for maps there are two types to be specified: type of keys and of values; we do it using angle brackets, as we will see in the examples below.

4.1 Collections

The Collection interface contains the following methods, which therefore must be implemented in all concrete classes representing collections:

- add — adds an element;
- addAll — adds all elements of another collection;
- clear — removes all elements;
- contains — checks if an object belongs to this collection;
- containsAll — checks if all elements of another collection belong to this collection;
- equals — compares this collection with another one;
- hashCode — return hash code of the collection (overriding the method inherited from Object);

- **empty** — returns **true** if this collection is empty;
- **iterator** — returns an iterator associated with this collection (this is an implementation of the interface **Iterable**);
- **parallelStream** — returns parallel stream with this collection as its source (defaulted);
- **remove** — removes an element from this collection;
- **removeAll** — removes all elements from another collection from this collection;
- **removeIf** — removes all elements satisfying a given predicate (defaulted);
- **retainAll** — retains in this collection only elements from another collection;
- **size** — returns the number of elements in this collection;
- **splitIterator** — returns a split iterator associated with this collection (defaulted);
- **stream** — returns the stream with this collection as its source;
- **toArray** — returns an array containing all elements of this collection;

Some collections do not permit certain operations — then their implementations just throw an exception; we say that these operations (represented by methods) are then *optional*.

There are some subinterfaces of **Collection** that are more specific for various kinds of collections. The most important are

- **List**: lists represent collections of elements that are ordered (like elements of an array) — it makes sense to talk about element number 0, number 1, etc. Therefore, there are some methods which are not applicable to all collections but specific to lists: e.g., **get(i)** which returns element with the index specified, **indexOf(val)** which returns the index of an element with a given value, **add(i,val)** which adds new element at the position specified, etc. For all lists, adding a new element at the end is very efficient; at other locations — not necessarily so. A list may contain equal values at different positions. There are two main implementations of lists (concrete classes, *not* interfaces):
  - **ArrayList** — implementation is based on arrays. Adding elements *not* at the end may be very inefficient but access to all elements (by index) is almost immediate.
  - **LinkedList** — implementation is based on doubly-linked lists. Adding and removing may be fast, but access is slower.
- **Set**: sets represent collections of *unique* elements (no two elements are equal). There are two main implementations of sets:
  - **TreeSet** — implementation is based on red-black tree; the elements have a well defined order (therefore they have to be *comparable*), access to elements is fast (logarithmic).
  - **HashSet** — implementation is based on hashing technique — access to elements is even faster (constant time) but the order is unspecified.

4.2 Maps

Maps represent sets of *pairs* of objects: the first element of a pair is a **key** and the second is a **value** associated with this key (as elements of an array are associated with their indices, which therefore may be viewed as playing the rôle of keys). Types of keys and values may be, and usually are, different. As keys are used as “indices”, there may be no two equal keys in any map — they have to be unique (but the same value may be associated with two different keys). All maps have to implement methods from Map interface; among others, these are:

- put(key, val) — adds a (key,value) pair to this map;
- putIfAbsent(key, val) — adds a pair to this map if the specified key is *not* already present (returning null); otherwise it does nothing and returns the value already associated with the key;
- get(key) — returns the value associated with a given key, or null if the map doesn't contain this key;
- getOrDefault(key, defaultVal) — returns the value associated with a given key, or defaultVal if the map doesn't contain this key;
- clear — removes all elements;
- containsKey — checks if there is a pair (the so called **entry**) in this map with a given key;
- containsValue — checks if there is a pair in this map with a given value (this is usually rather inefficient);
- size — returns the number of elements (entries) in this map;
- isEmpty — checks if the map is empty;
- remove(key) — removes the entry with a given key;
- keySet — returning a set of all keys;
- values — returning a collection of all values;
- entrySet — returning set of entries, each of which has a key and a value accessible by methods getKey and geValue;
- etc.

There are two main implementations of maps:

- TreeMap — implementation is based on red-black tree of *keys*; the elements must have a well defined order. Therefore, *keys* have to be Comparable, or you have to pass a Comparator to the constructor. Access to elements is fast (logarithmic).
- HashMap — implementation is based on hashing technique — access to elements is even faster (constant time) but the order is unspecified.

4.3 Iterators

Iterators are objects representing a "view" of the elements of a collection and can yield on demand these elements, remembering which have already been yielded and knowing if there is anything that has not been returned yet. Iterators are also "generic", so we should always specify the type of elements which they will return. The Iterator interface declares methods (here T is the type of elements)

```
        boolean hasNext()
        T next()
```

The first returns the next element from those that have not been returned yet. The second tells us if there is still an element that has not been returned by next (calling next when hasNext returns false triggers the exception NoSuchElementException). There is the third operation which can be invoked on an iterators, remove: it removes the last element of the underlying collection that has just been returned by next. However, we don't have to implement it, because this operation is optional and it already *has* a default implementation (which just throws an exception.) There is also a default (already implemented) method forEachRemaining.

The collection of elements which backs an iterator can be anything that implements **Iterable** interface. All “real” collections — classes from the standard library that implement **Collection** — *do* implement **Iterable**. **Iterable** declares just one abstract method:

```
Iterator<T> iterator()
```

(where **T** is the type of elements) which returns an iterator associated with a given collection implementing **Iterable**. In fact, this need not to be a “real” collection, it should only behave as one from the point of view of the iterator returned.

Let us consider an example — here object of type **IterableRange** plays the rôle of a collection, although there is no array or any other “true” collection involved: however, it implements **Iterable** and the iterator returned by its **iterator** method meets all the requirements of an iterator:

Listing 49 JIX-RangIter/RangIter.java

```java
import java.util.Iterator;
import java.util.NoSuchElementException;

  // main class for testing
public class RangIter {
    public static void main (String[] args) {
        Iterable<Integer> iterab1 = new IterableRange(3,7);
        Iterator<Integer> iter = iterab1.iterator();
        while (iter.hasNext())
            System.out.print(iter.next() + " ");

        System.out.println("\nand now foreach:");
        for (Integer i : new IterableRange(3,7))
            System.out.print(i + " ");
        System.out.println();
    }
}

  // objects of this class are 'iterable', i.e., they
  // behave (at least to some extend) as collections
class IterableRange implements Iterable<Integer> {
    private int a, b;
    IterableRange(int a, int b) {
        this.a = a;
        this.b = b;
    }
    public Iterator<Integer> iterator() {
        return new RangeIterator(a,b);
    }
}

  // object of this class bahave like interators
  // traversing the IterableRange 'collections'
class RangeIterator implements Iterator<Integer> {
    private int a, b;
    private int curr;
    RangeIterator(int a, int b) {
        this.a = a;
        this.b = b;
        curr = a;
    }
    @Override
    public boolean hasNext() {
        return curr <= b;
    }
    @Override
    public Integer next() {
        if (!hasNext()) throw new NoSuchElementException();
        return curr++;
    }
    // since Java 1.8 remove has a default implementation
}
```

All classes are here defined in one file for simplicity, it is generally not a recommended practice. Note that the class **RangeIterator** could have been defined inside **IterableRange** (see sec. 1.3). Note also the *for-each* (called also *ranged for*) loop used here in the **main** function:

```
for (Integer i : new IterableRange(3, 7))
    System.out.print(i + " ");
```

This form of loops works not only with standard collections, but in fact with anything that implements **Iterable**. Basically, the form

```
for (Type elem : iterable_object)
    do_something_with_elem
```

is by the compiler automatically translated into something like this

```
{
    Iterator<Type> it = iterable_object.iterator();
    while (it.hasNext()) {
        Type elem = it.next();
        do_something_with_elem
    }
}
```

4.4 *Examples*

Let us consider a few examples. First, lists and sets (i.e., collections implementing the **Collection** interface)

Listing 50 HUG-Lists/AList.java

```
import java.util.ArrayList;
import java.util.Collection;
import java.util.Collections;
import java.util.List;
import java.util.Set;
import java.util.HashSet;
import java.util.TreeSet;
import java.util.Iterator;

public class AList {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("Sue");
        list.add("Lea");
        list.add("Ann");
        list.add("Kim");
        list.add("Lea");
        list.add(1,"Amy");  // inefficient!

        System.out.println("First 'Lea' under index " +
                                   list.indexOf("Lea"));
        System.out.println("Last  'Lea' under index " +
                               list.lastIndexOf("Lea"));
        System.out.println("Does the list contain 'Sue': " +
                                     list.contains("Sue"));
        System.out.println("Does the list contain 'Bea': " +
                                     list.contains("Bea"));
        System.out.println("Size of list: " + list.size());

          // traditional looping
        System.out.print("With get():  ");
        for (int i = 0; i < list.size(); ++i)
            System.out.print(" " + list.get(i));
          // `foreach' loop
        System.out.print("\nAnother way: ");
        for (String s : list)
            System.out.print(" " + s);
          // iterators
        System.out.print("\nnow iterator:");
        Iterator<String> iter = list.iterator();
        while (iter.hasNext())
            System.out.print(" " + iter.next());

          // sorting and sublists
        Collections.sort(list);
        System.out.println("\nSorted:      " + list);
        List<String> subl = list.subList(1,4);
        System.out.println("Sublist:     " + subl);

          // HashSet
        Set<String> hSet = new HashSet<>(list);
        System.out.println("Hash set:    " + hSet);
```

```
          // TreeSet
        Set<String> tSet = new TreeSet<>(list);
        System.out.println("Tree set:    " + tSet);
        System.out.println("Does the tSet contain 'Sue': " +
                                     tSet  .contains("Sue"));
        tSet.add("Zoe");
        tSet.remove("Lea");
        System.out.println("tSet now:    " + tSet);
    }
}
```

The program prints

```
First 'Lea' under index 2
Last  'Lea' under index 5
Does the list contain 'Sue': true
Does the list contain 'Bea': false
Size of list: 6
With get():   Sue Amy Lea Ann Kim Lea
Another way:  Sue Amy Lea Ann Kim Lea
now iterator: Sue Amy Lea Ann Kim Lea
Sorted:      [Amy, Ann, Kim, Lea, Lea, Sue]
Sublist:     [Ann, Kim, Lea]
Hash set:    [Ann, Sue, Lea, Amy, Kim]
Tree set:    [Amy, Ann, Kim, Lea, Sue]
Does the tSet contain 'Sue': true
tSet now:    [Amy, Ann, Kim, Sue, Zoe]
```

Note, that iteration over a **TreeSet** collection gives its elements in alphabetical order, what corresponds to the natural order of **String**s. This is not true for a **HashSet**. Note also various forms of iterations over **List**s, in particular the *for-each* loop (that we already used for arrays).

Now maps (i.e., collections implementing the **Map** interface, but not **Collection**):

Listing 51 HUK-SimpleMap/ASimpleMap.java

```
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;
import java.util.TreeMap;

public class ASimpleMap {
    public static void main(String[] args) {
        Map<String,Integer> map = new TreeMap<>();
        map.put("Sue",167);
        map.put("Ann",173);
        map.put("Lea",170);
        map.put("Kim",173);

```

```java
        Iterator<String> iter = map.keySet().iterator();
        while (iter.hasNext()) {
            String key = iter.next();
            int    val = map.get(key); // auto(un)boxing
            System.out.print(key + ": " + val + "  ");
            if (val == 170) iter.remove();
        }

        System.out.print("\nWe can print a map: " + map);

          // returns null, or the value if present
        map.putIfAbsent("Lea",169);
          // returns old value or null if not present
        map.replace("Lea",170);

        System.out.print("\nIs there a key 'Lea'? " +
                              map.containsKey("Lea"));
        System.out.print("\nIs there a key 'Zoe'? " +
                              map.containsKey("Zoe"));
          // inefficient!
        System.out.println("\nIs any girl 170 cm tall? " +
                                   map.containsValue(170));

        Integer was = map.remove("Ann");
        if (was != null)
            System.out.println("Ann removed, she was " +
                was + " cm tall");
        was = map.remove("Zoe");
        if (was == null)
            System.out.println("There was no 'Zoe'!");

        System.out.println("getOrDefault: 'Zoe'->" +
                         map.getOrDefault("Zoe",-1));
        System.out.println("getOrDefault: 'Lea'->" +
                         map.getOrDefault("Lea",-1));

        System.out.print("Iterating over 'keySet': ");
        for (String s : map.keySet())
            System.out.print(s + "->" + map.get(s) +"  ");

        System.out.print("\nMore efficient way: ");
        for (Map.Entry<String,Integer> e : map.entrySet())
            System.out.print(e.getKey() + "->" +
                             e.getValue() + "  ");
        System.out.println();
    }
}
```

The program prints

```
Ann: 173  Kim: 173  Lea: 170  Sue: 167
We can print a map: {Ann=173, Kim=173, Sue=167}
Is there a key 'Lea'? true
Is there a key 'Zoe'? false
Is any girl 170 cm tall? true
Ann removed, she was 173 cm tall
There was no 'Zoe'!
getOrDefault: 'Zoe'->-1
getOrDefault: 'Lea'->170
Iterating over 'keySet': Kim->173  Lea->170  Sue->167
More efficient way: Kim->173  Lea->170  Sue->167
```

Note, that Maps themselves are *not* iterable; however, a map's key set (returned by the keySet method), being a Set, i.e., a Collection, *is* iterable. The same applies to the Set of type Map.Entry objects returned by the entrySet method — each of such objects represents one (key, value) pair, of which both components may be accessed separately by the methods getKey and getValue.

4.5 Importance of equal and hashCode methods

Class Object defines — among others (in particular

```
    public String toString()
```

that we already know) two other important methods:

- `public boolean equals(Object ob)` — which is used to check if two objects are, according to some criterion, equal: for two references `ob1` and `ob2`, the expression `ob1.equals(ob2)` compares the objects and yields **true** or **false**. But according to what criterion? In class Object there is no data to be compared, so the default implementation just compares addresses of the two objects. Very often, this is *not* what we want. When we have two objects of class Person and these persons have identical names, dates of birth, passport numbers etc., we rather want to consider the two objects as representing exactly the same person, in other words we want to consider these two objects equal. To get this behavior, we thus have to redefine (override) equals in our class.
  The equals method should implement an equivalence relation on non-null object references. This means that for any non-null references `a`, `b` and `c` `a.equals(a)` should always be **true** (reflexivity), `a.equals(b)` should have the same value (**true** or **false**) as `b.equals(a)` (symmetry), and if `a.equals(b)` and `b.equals(c)` are **true** then `a.equals(c)` must also be **true** (transitivity).
  Reflexivity and symmetry are usually obvious, but transitivity — not always. Suppose, we consider two two-element sets equal if they have at least one common element. Then $A = \{a, b\}$ is equal to $B = \{b, c\}$ and B is equal to $C = \{c, d\}$, but A and C have no common element and are not equal.
- `public int hashCode()` — which is used to calculate the so called **hash code** of an object. This is necessary if we want to put objects of our class in collections implemented as hash tables (e.g., HashSet or keys in a HashMap). The implementation of hashCode method from the Object class uses just the address of the object to calculate its hash code. However, very often this is not desirable as it would lead to situations when two objects that are considered equal (according to equals) have different hash codes: as a consequence the collections of such objects would be invalidated. Therefore, we have to override also this method

remembering to do it consistently: whenever two objects are equal according to **equal**, their hash codes should be exactly the same.

This is illustrated by the following example:

Listing 52 HUM-HashEquals/Person.java

```java
public class Person {

    private String name;
    private String idNumber;

    public Person(String name, String idNumber) {
        this.name     = name;
        this.idNumber = idNumber;
    }

    /**/
    @Override
    public boolean equals(Object other) {
        if (other == null ||
            getClass() != other.getClass()) return false;
        Person p = (Person)other;
        return idNumber.equals(p.idNumber) &&
                     name.equals(p.name);
    }
    /**/

    /**/
    @Override
    public int hashCode() {
        return 17*name.hashCode() + idNumber.hashCode();
    }
    /**/

    @Override
    public String toString() {
        return name + "(" + idNumber + ")";
    }
}
```

with **main** as below

Listing 53 HUM-HashEquals/AHash.java

```java
import java.util.HashMap;
import java.util.Map;

public class AHash {
    public static void main(String[] args) {
        Map<Person,String> map = new HashMap<>();

        map.put(new Person("Sue","123456"),"Sue");

          // new object, but should be equivalent to
          // the one which has been put into the map
        Person sue = new Person("Sue","123456");

        if (map.containsKey(sue))
            System.out.println(sue + " has been found");
        else
            System.out.println(sue + " has NOT been found");
    }
}
```

As can be easily checked, the program will print

```
Sue(123456) has been found
```

only when both **hashCode** and **equals** are consistently overridden in the class of keys of the map (i.e., **Person**).

Section 5

Method references

Quite often, when we override a method of a functional interface, its implementation reduces to invoking another method that already exists in a class. In such situations we can pass the reference to this method and the compiler will do the rest all by itself. For example, suppose we have a stream stream of objects of type **AClass**; we can terminate the pipeline of operations on the stream with **forEach** which then expects an object implementing **Consumer<AClass>**. Suppose, we just want to print elements of the stream — we can achieve this by writing

```
    stream.forEach(e -> System.out.println(e))
```

or, in an abbreviated form, we just pass a method reference

```
    stream.forEach(System.out::println)
```

In the latter case, we are telling the compiler *take elements of the stream and pass them, one by one, to the method indicated as an argument.* Notice, that in a method reference the name of the method must be preceded by a double colon and there are no parentheses after the name, because we don't invoke this function here; it is just a reference to the method itself. As **println** is a non-static method, it must be called on an object, so in front of the method reference we specify the object it is to be invoked on (in this case it is System.out).

There are three main forms of method references:

- anObject::nonstaticMethod
- AClass::staticMethod
- AClass::nonstaticMethod

In each case arguments for methods are needed, and also, in the third case, an object which will be the receiver of the invocation.

In the first case, anObject::nonstaticMethod, arguments will be passed as the argument to the method, so it is essentially equivalent to a lambda

```
    e -> anObject.nonstaticMethod(e)
    (e,f) -> anObject.nonstaticMethod(e,f)
```

depending of the number of arguments expected. Note that there can be several overloaded versions of the method: compiler will select the one matching the number and types of arguments passed. We have already seen an example: System.out::println corresponds to

```
    e -> System.out.println(e)
```

In the second case, we can use a static method if the number and type of arguments matches the number and type of arguments of the static method. For example, if **Function<Double,Double,Double>** (or **BinaryOperator<Double>**) is expected in a given context, we can pass just Math::pow; this will be equivalent to passing the the lambda

```
    (x, y) -> Math.pow(x,y)
```

In the third case, the first argument becomes the target (receiver) of the method and the remaining arguments are passed to the method as arguments. For example, String::compareToIgnoreCase corresponds to

```
(x, y) -> x.compareToIgnoreCase(y).
```

Suppose you want to sort an array of Strings (say, strings) ignoring the case. You can call sort which expects a Comparator that will be called with two arguments; you can pass the method reference

```
Arrays.sort(strings, String::compareToIgnoreCase)
```

Let us consider an example:

Listing 54 LCP-MethRefs/MethRefs.java

```
import java.util.Arrays;
import java.util.Collections;
import java.util.Iterator;
import java.util.List;

public class MethRefs {
    List<String> list = Arrays.asList(
                "Zoe", "kate", "Cindy", "barbra");
    public static void main(String[] args) {
        new MethRefs();
    }

    public MethRefs() {
          // case anObject::nonstaticMethod
          // Iterable<String> expected
        Iterable<String> iterObj = this::getIter;
        for (String s : iterObj) System.out.print(s + " ");
        System.out.println();

          // case AClass::staticMethod
          // Runnable expected
        Thread t = new Thread(MethRefs::tenFibos);
        t.start();
        try {
            t.join();
        } catch(InterruptedException ignore) { }

          // case AClass::nonstaticMethod
          // Comparator<String> expected
        Collections.sort(list, String::compareTo);
        System.out.println(list);
        Collections.sort(list, String::compareToIgnoreCase);
        System.out.println(list);
    }

    public static void tenFibos() {
          // prints first 10 Fibonacci numbers
        StringBuilder sb = new StringBuilder("0, 1");
        int a = 0, b = 1;
        for (int i = 0; i < 8; ++i) {
            b += a;
            a  = b - a;
            sb.append(", " + b);
        }
        System.out.println(sb);
    }

    public Iterator<String> getIter() {
        return list.iterator();
    }
}
```

which prints

```
Zoe kate Cindy barbra
0, 1, 1, 2, 3, 5, 8, 13, 21, 34
[Cindy, Zoe, barbra, kate]
[barbra, Cindy, kate, Zoe]
```

More examples can be found in Listing 56, Listing 58, Listing 59 and other listings from section 6 on p. 64.

Method references may also refer to constructors, even ‘constructors’ of arrays. The syntax is as follows:

- AClass::new
- AType[]::new

where in the second case **AType** may also be a primitive type (like **int**). In the first case, arguments are passed to a constructor, and its type decides which constructor will be used (if they are overloaded).
In the second case, there can be only one argument, a non-negative integer, which will be used as the size of the array created. There are also functions that accept a reference to an array constructor specifying type of the array to be created, which would otherwise be **Object[]**. For examples — seeListing 55 and Listing 62.

Section 6

Streams

6.1 Introduction

Streams provide an abstraction for dealing with collections of values (which do not have to correspond to 'real' collections) and specifying what you want to be done, leaving the details of how to do it to the library. For example, you can create a stream of object of type **Person** and say something like *select only women of age in the range* [35, 60] *and form a map with country of origin as keys and lists of women form these countries as values.* Or, having an infinite (*sic!*) stream of **double**s, you can say *select only positive numbers, round them to integers, calculate their squares, take 1000 first elements and give me their arithmetic average.* In addition, you can often say something like *take my stream, do this and this and try to do it using as many separate threads as necessary to ensure maximum efficiency.*

Streams can be created from collections, arrays, or using generators or iterators. However, themselves, they are *not* collections. Rather, they represent streams of individual pieces of data that can be processed one by one, not necessarily storing anywhere and anytime all of them together.

- Usually, streams do not store their elements anywhere, they take it one by one from a source. However, for some operations they have to store the elements internally, for example in order to sort them.
- Streams do not modify the source of their elements, they can transform them and yield transformed values in a new stream.
- Operations on streams are 'lazy', i.e. only those operation that are needed to get the desired result are actually executed.

In order to use streams, you have to

- create a stream — you can require it to be parallelized if it makes sense for a problem at hand;
- apply zero, one or more *intermediate operations*; each of them yields another stream to which one can apply a next intermediate operation, thus forming the so called *pipeline*;
- apply a *terminal operation* that produces a result. Only a terminal operation triggers all the other (intermediate) operations; before that no processing takes place. This is why streams can be lazy — when it is known what result is expected, it is known which operations are necessary and which are not. After applying a terminal operation, the stream is deemed 'consumed'; its is closed and cannot be reused.

Some operations, both intermediate and terminal, can be *short-circuiting*: an intermediate operation is short-circuiting if for an infinite stream it may produce a finite stream, a terminal operation is short-circuiting if for an infinite stream it may terminate in finite time. Of course, for infinite streams, there must be at least one short-circuiting operation in the whole pipeline of operations.

Intermediate operations can be stateless or stateful. Most of them are stateless — elements are processed one by one and to process a given element no knowledge of elements seen previously, or those to be seen later, is needed. Some operations, however, are stateful — at least some information on the previous elements has to be remembered. There are four such operations in the library: **sorted**, **distinct**, **limit** and **skip**. For obvious reasons, the stateful operations cannot be applied to infinite streams and are less efficient.

In the examples below, we will use references to methods, which can be used instead of lambdas — they are covered in more detail in sec. 5 on p. 61.
We will also refer to functional interfaces that are already defined in the library: more about them in sec. 7 on p. 80.

6.2 Creating streams

Streams can be created from any collection by invoking their **stream** method

```
collection.stream()
```

which produces a stream of type **Stream<T>**, where **T** is the type of elements of the collection. There is also a static function **of** in class **Stream**. It takes any number of arguments, or an array, and creates a stream:

```
Stream<String> s1 = Stream.of("Alice", "Cindy", "Kate");
Stream<String> s2 = Stream.of(new String[]{"A","B"});
```

Another static function of **Stream** is **generate**. It takes a **Supplier** and produces a stream by using the supplier to create (potentially infinite) stream of values

```
Stream<Double> d = Stream.generate(Math::random);
```

One can also use **iterate** which takes a 'seed' and a **UnaryOperator** and repeatedly applies the operator to the previous element with seed as the first one, so the resulting stream will be
$\quad$(seed, op.apply(seed), op.apply(op.apply(seed)), $\ldots$)
For example:

```
Stream.iterate(1, n -> 2*n).limit(7).forEach(System.out::println);
```

will print the sequence $(1, 2, 4, 8, 16, 32, 64)$; note that we had to limit the number of elements in the stream, as **iterate** generates potentially infinite sequence of numbers.

Functions **generate** and **iterate** may also be used with primitive types; the type of the stream will then be **DoubleStream**, **LongStream** or **IntStream**.

Streams of primitive type, **IntStream** and **LongStream** can also be created by calling **range** and **rangeClosed**:

```
IntStream s = IntStream.range(int startIncl, int endExcl)
IntStream s = IntStream.rangeClosed(int startIncl, int endIncl)
```

which return an ordered stream of **int**s from startIncl (inclusive) to endExcl (exclusive) for the first form or to endIncl (inclusive) in the second form, by an incremental step of 1 (and analogously for **long**s).

Methods creating streams have also been added to some other classes, for example **Files** from ***java.nio.file***. Its static method **lines** yields a stream of lines of a file as **String**s:

```
try (Stream<String> str = Files.lines(path,charset)) {
    // str is a stream of lines as strings
}
```

We use try-with-resources here to be sure that when the stream is closed, the underlying file is also closed (if your charset is UFT-8, it can be omitted, as this is the default).

There is also, in class **Pattern** from ***java.util.regex***, the method **splitAsStream** which yields a stream of **String**s resulting from splitting a text (actually, a **CharSequence**) with a given regex specifying the separator:

```
Stream<String> words =
    Pattern.compile("\\P{L}+").splitAsStream(text);
```

will yield a stream of words from text (separated by non-empty sequences of non-letters).

You can find examples of stream creation in listings that follow (in particular, Listing 57 on page 71).

6.3 Intermediate operations

Let us enumerate the most useful intermediate operations (implemented as methods of class **Stream** from the package ***java.util.stream***). In what follows, the symbol **T** denotes the type of elements in a stream on which the methods are invoked (its type is therefore **Stream<T>**, while **R** stands for another type, where applicable.

- `Stream<T> filter(Predicate<T> pred)` — produces a new stream where only elements that satisfy the predicate from the original stream are retained. Examples: Listing 56, Listing 60, Listing 61, Listing 63.
- `Stream<R> map(Function<T,R> fun)` — applies the function to each element of the input stream and produces a new stream of values obtained. There are similar operations with a primitive type instead of **R**: **mapToInt**, **mapToLong** and **mapToDouble**. Examples: Listing 55, Listing 56, Listing 57, Listing 58, Listing 61, Listing 62.
- `Stream<T> distinct()` — produces the same stream as the input one but with all duplicates removed (they need not to be on subsequent positions). To compare elements, method **equal** is used. Stateful (and rather expensive) operation. Examples: Listing 57, Listing 63.
- `Stream<R> flatMap(Function<T,Stream<R>> fun)` — applies **fun** to all elements to get stream of streams; then 'flattens' these streams into one stream of all elements from the individual streams. There are specialized versions for primitive types instead of **R**: **flatMapToInt**, **flatMapToLong** and **flatMapToDouble** which return **IntStream**, **LongStream** and **DoubleStream**, respectively. Example: Listing 63.
- `Stream<T> sorted(Comparator<T> cmp)` — yields the same stream but sorted by using the given comparator cmp. There is also version without any arguments — then the natural order of type **T** is used. Stateful operation. See also the documentation of **Comparator** from ***java.util*** for various ways of creating comparators; an example is shown in Listing 63. Other examples include Listing 57 and Listing 61.
- `Stream<T> peek(Consumer<T> cons)` — returns the same stream but executes **cons** on each element 'on the fly'. Used often for debugging. Examples: Listing 61, Listing 55, Listing 56.

- `Stream<T> limit(long lim)` — returns the same stream but limited to at most lim first elements. Stateful, short-circuiting operation. Example: Listing 55.
- `Stream<T> skip(long skp)` — returns the same stream but with first `skp` elements suppressed. Stateful operation.

6.4 Terminal operations

Zero, one or more transformations by intermediate operations must be followed by exactly one final (terminal) operation; otherwise no operation at all would be actually executed as this is a terminal operation that forces the execution of all (lazy) operations that precede it. Terminal operation yields a result — this can be one value (in particular, a number) or a collection (or array) of elements.

Many terminal operations can be created by stream method **collect(Collector)**, where a collector can be chosen from the rich set of ready-to-use collectors returned by static factory methods defined in class **Collectors**.

There are several terminal operations yielding a number as their result. For example (**T** denotes the type of elements in the stream, **NumType** stands for **Int**, **Long** or **Double**):

- `long count()` — returns the number of elements in the input stream;
- `Optional<T> min(Comparator cmp)`, `Optional<T> max(Comparator cmp)` — return minimum and maximum elements of the stream using a provided comparator. For primitive-type streams no comparator is needed and the return type is **OptionalInt**, **OptionalLong** or **OptionalDouble**. To get minimum or maximum element, one can also use **collect** with a collector returned by invoking **minBy(Comparator)** or **maxBy(Comparator)** from class **Collectors**. The optional returned is empty, if stream was empty.
- `NumType sum()` — returns the sum of elements for primitive-type streams. Similarly, a collector returned by **summingType(ToTypeFunction)** can be used for non-primitive type (**Type** is **Int**, **Long** or **Double**). See examples in Listing 57 and Listing 61.
- `Optional<Double> average()` — returns the mean (arithmetic average) for streams of primitive types. A collector returned by **averagingType(ToTypeFunction)** can be used for non-primitive types (**Type** is **Int**, **Long** or **Double**).

For primitive streams, there are also terminal operations **summaryStatistics**, which return an object of type **TypeSummaryStatictics**, where **Type** is **Int**, **Long** or **Double**. This object may then be queried for the number of elements, their sum, average, minimum and maximum. For non-primitive types one can use a collector returned by **summarizingType(ToTypeFunction)**. Very often, the **ToTypeFunction** function will be just the reference to a method returning a number. See an example in Listing 58.

Another group of terminal operations are those returning collections (or arrays). Let us mention some of them

- `Object[] toArray()` — returns all elements of the stream as an array (of type **Object[]**). See example in Listing 62.
- `R[] toArray(IntFunction<R[]> gen)` — returns an array of type **R[]**; `gen` is a function taking an **int** and creating an array of this size. Most often this will be just the reference to an array constructor **R[]::new**. See examples in Listing 56 and Listing 60.

- collector retuned by `groupingBy(Function<T,K> keyExtr)` — produces a map of type **Map<K,List<T>>**. It applies the function **keyExtr** to each element of the stream and treats the obtained value as a key in the resulting map; lists of elements yielding the same key will be the values of the map (there are others, very powerful versions of this collector). See a simple example in Listing 59.

A very useful terminal operation just 'consumes' the stream

```
void forEach(Consumer<T> cons)
```

by invoking cons on each element; a very common example is just printing:
`stream.forEach(System.out::println).`
One can also create a **String** by concatenating strings obtained from subsequent elements of a stream; such a collector can be created by factory method

```
String stream.collect(Collectors.joining())
```

(a desired delimiter may also be passed as the argument). See examples in Listing 57, Listing 58, Listing 61 and Listing 63.

There are also short-circuiting operations taking a **Predicate** and returning **boolean** — **allMatch** (whether all elements satisfy the given predicate), **anyMatch** (whether at least one satisfies it) and **noneMatch** (whether all elements do not satisfy the predicate). They are short-circuiting because the stream is deemed consumed (and processing stops) when the result is already known. For example, in the case of **allMatch**, when an element *not* satisfying the predicate is encountered, the answer is **false** and cannot change, so the processing stops. See example in Listing 57.

A very important and versatile reduction of a stream to a single value can be obtained by invoking **reduce** (the operation performed by **reduce** is called *left fold*). The operation has a few variants. The basic one looks like this

```
T reduce(T iden, BinaryOperator<T> acc)
```

Here, acc is a **BinaryOperator<T>** acting on two values of type **T** (type of elements of the stream). First the operator is applied to iden and the first element of the stream, then on the result and the second element, than again on the result and the third, and so on. Basically, this is equivalent to

```
T result = iden;
for (T e : elements of the stream)
    result = acc.apply(result, e)
return result;
```

There are two important conditions, though:

- iden must be an identity of the operation, i.e.,
  `apply(iden,e)`
  must return e (like number 0 for addition or 1 for multiplication);
- operator **apply** must be associative, i.e.,
  `apply(apply(e,f),g) = apply(e,apply(f,g))`
  must always hold.

Let us suppose that we have a stream str of **Integers**. Then we could use **reduce** to find the sum or product of all elements, or their count, like this

```
  // sum
Integer a = str.reduce(0, Integer::sum);
  // product
Integer a = str.reduce(1, (a,e) -> a*e);
  // count
Integer a = str.reduce(0, (a,e) -> a+1);
```

There are also other versions of **reduce** (without iden, but returning **Optional** — see documentation).

Streams can be parallelized. You either get a parallel stream directly from a collection

```
Stream<T> parellelStream = collection.parallelStream();
```

or parallelize an existing stream

```
Stream<T> parallelStream = sequentialStream.parallel();
```

You can also make a parallel stream sequential

```
Stream<T> sequentialStream = parallelStream.sequential();
```

Parallel streams allow the compiler to divide operations on the stream into parts that can be executed on different threads. Of course, the final result will have to be somehow combined from the partial results — this is not always trivial! It is your responsibility to ensure that operations performed on different threads do not lead to data races or deadlocks.

6.5 Examples

The example below uses **limit**, **peek**, **map** and **forEach** terminal operation. It demonstrates the 'laziness' of streams: only those elements which are necessary to get the result are actually processed. Also, a reference to a constructor is used:

Listing 55 LDC-Lazy/Lazy.java

```java
import java.util.stream.Stream;
import java.io.File;

public class Lazy {
    public static void main (String[] args) {
        Stream.of("Alice", "Bella", "Cecilia", "Dorothy")
          .peek(e -> System.out.print("peek: " + e + "; "))
          .map(s -> s + ".txt")
          .map(File::new)
          .limit(2)
          .forEach(f -> System.out.println(f + " exists? " +
                                  (f.exists() ? "Yes" : "No") ));
    }
}
```

The output is interesting:

```
peek: Alice; Alice.txt exists? No
peek: Bella; Bella.txt exists? Yes
```

As we can see, the first element ("Alice") started its "journey" and went all the way down to the **forEach** before the next element ("Bella") even started! After two first elements ("Alice" and "Bella") finished, the **limit(2)** "says" *that's enough* and no other element even starts its journey along the chain of operations — ("Cecilia" and "Dorothy" don't even reach **peek**! This nicely illustrates the laziness of streams: only what is necessary to get the final result will be really executed.

The example below demonstrates the function **Files.lines** which creates a stream of lines of a given file which then can be transformed and reduced to a desired result (in this case, a list). It also demonstrates method references, as well as methods **filter**, **peek**, **map** and **toList** and **forEach** terminal operations.

Listing 56 LDF-StreamGrep/StreamGrep.java

```java
import java.util.List;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.stream.Collectors;
import java.util.stream.Stream;
import static java.nio.charset.StandardCharsets.UTF_8;

public class StreamGrep {
    public static void main(String[] args) {
        List<String> list = null;
        try (Stream<String> lines =
                Files.lines(Paths.get("StreamGrep.java"),
                            UTF_8)) {
            String substr = "String";
            list = lines
                       // Predicate expected
                     .filter(s -> s.indexOf(substr) >= 0)
                       // Consumer expected
                     .peek(System.out::println)
                     .collect(Collectors.toList());
        } catch(IOException e) { return; }
        System.out.println("and now the list...");
        list.stream()
                // reference to method
              .map(String::toUpperCase)
              .forEach(System.out::println);
    }
}
```

The program prints

```
public static void main(String[] args) {
    List<String> list = null;
```

```
            try (Stream<String> lines =
                String substr = "String";
                .map(String::toUpperCase)
    and now the list...
        PUBLIC STATIC VOID MAIN(STRING[] ARGS) {
            LIST<STRING> LIST = NULL;
            TRY (STREAM<STRING> LINES =
                STRING SUBSTR = "STRING";
                .MAP(STRING::TOUPPERCASE)
```

A stream of lines of a file can also be obtained from **BufferedReader**, as the example below demonstrates. It also shows streams of primitive type (e.g., **int**), various ways of creating streams, functions **sorted**, **distinct**, **filter**, **map**, **mapToInt**, **peek** and **sum**, **joining**, **allMatch** and **forEach** terminal operations. The last part of the program shows how to obtain a stream from a regex.

Listing 57 LDD-StreamMisc/StreamMisc.java

```java
import java.io.BufferedReader;
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.IntStream;
import java.util.stream.Stream;
import java.util.regex.Pattern;

public class StreamMisc {
    public static void main(String[] args) {
        System.out.println("*From an arrray...");
        String[] ws = {"To", "be", "or", "not", "to", "be"};
        Stream.of(ws)
         .map(String::toLowerCase)
         .distinct()
         .sorted()
         .forEach(e -> System.out.print(e + " "));
        System.out.println();

        System.out.println("*From varargs...");
        System.out.println(
            Stream.of("To", "be", "or", "not", "to", "be")
             .collect(Collectors.joining(" - "))
        );

        System.out.println("*From a collection...");
        List<String> list = Arrays.asList(
                    "1","10","100","1000","10000","100000");
        System.out.println("Sum = " +
            list.stream().mapToInt(Integer::parseInt).sum()
        );

          // generating a stream by iterating a unary
          // function starting from a~given seed
        System.out.println("*From a generator...");
        ArrayList<Integer> arri = new ArrayList<>();
        IntStream.iterate(17, n -> n%2 == 0 ? n/2 : 3*n+1)
         .peek(arri::add)         // arri in closure,
         .allMatch(n -> n != 1);  // allMatch is short-
                                  // circuited, so will
        System.out.println(arri); // stop iteration!

        System.out.println("*Lines of a file as stream...");
        try (BufferedReader br = Files.newBufferedReader(
                    Paths.get("StreamMisc.java"),
                    StandardCharsets.UTF_8)) // default
        {
            br.lines()
             .filter(e -> e.contains("collect"))
             .forEach(System.out::println);
        } catch (IOException never_ignore_exceptions) { }

        System.out.println("*From a regex...");
        String s = "a is 1, b=3 and c:7 X";
        System.out.println("Sum of extracted numbers is " +
                Pattern.compile("\\D+")
                 .splitAsStream(s)
                 .filter(e -> e.length() > 0)
                 .mapToInt(Integer::parseInt).sum());
    }
}
```

The program prints

```
*From an arrray...
be not or to
*From varargs...
To - be - or - not - to - be
*From a collection...
Sum = 111111
*From a generator...
[17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1]
*Lines of a file as stream...
             .collect(Collectors.joining(" - "))
        System.out.println("*From a collection...");
             .filter(e -> e.contains("collect"))
*From a regex...
```

```
Sum of extracted numbers is 11
```

Primitive-type streams are also demonstrated in the example below; note the **summaryStatistics** terminal operation and the **generate** supplier. Also, functions **sorted**, **limit** and **map** are used here (as well as method references).

Listing 58 LDG-MethRefStr/MethRefStr.java

```java
import java.util.Random;
import java.util.stream.DoubleStream;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class MethRefStr {
    public static void main(String[] args) {
        Random r = new Random(); // effectively final
        System.out.println(
                  // Supplier of double's expected
                DoubleStream.generate(r::nextGaussian)
                  // we want just ten million numbers
               .limit(10_000_000)
                  // reduction to DoubleSummaryStatistics
               .summaryStatistics()
                  // arithmetic average of all numbers
               .getAverage());

        System.out.println(
            Stream.of(new Person("C"), new Person("A"),
                      new Person("D"), new Person("B"))
                     // Function<Person,otherType> expected
                  .map(Person::getName)
                  .sorted()
                     // Function<String,otherType> expected
                  .map(String::toLowerCase)
                     // reduction to a single String
                  .collect(Collectors.joining("-")));

        Thread t = new Thread(MethRefStr::fibos);
        t.start();
        try {
            t.join();
        } catch(InterruptedException ignore) { }
    }

    public static void fibos() {
        StringBuilder sb = new StringBuilder("0, 1");
        int a = 0, b = 1;
        for (int i = 0; i < 8; ++i) {
            b += a;
            a  = b - a;
            sb.append(", " + b);
        }
        System.out.println(sb);
    }
}

class Person {
    private String name;
    public Person(String n) { name = n; }
    public String getName() { return name; }
}
```

The program prints (the first number may be different)

```
-1.9002508699231762E-4
a-b-c-d
0, 1, 1, 2, 3, 5, 8, 13, 21, 34
```

The next example demonstrates the use of, extremely useful, **groupingBy** terminal operation. It comes in many variants; below, the simplest of them is used. It takes a function, which applied to elements of the stream will yield a value which will be then used as the key of the map: values of this map will be lists of elements that yield this key:

Listing 59 LDE-Grouping/Grouping.java

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class Grouping {
    public static void main (String[] args) {
        List<Person> list = Arrays.asList(
                new Person("John",  "UK"),
                new Person("Mary",  "US"),
                new Person("Xue",   "CH"),
                new Person("Kate",  "UK"),
                new Person("Janek", "PL"),
                new Person("Cindy", "US"),
                new Person("Bao",   "CH"),
                new Person("Kasia", "PL")
        );

          // collect gives Map<String,List<Person>>
          // groupingBy expects Function...
        list
         .stream()
         .collect(Collectors.groupingBy(Person::getCountry))
         .entrySet()
         .stream()
         .forEach(e -> System.out.println(e.getKey() +
                                 " -> " + e.getValue()));
    }
}

class Person {
    private final String name;
    private final String country;
    public Person(String n, String c) {
        name = n; country = c;
    }
    public String getName()    { return name;    }
    public String getCountry() { return country; }
    @Override
    public String toString() {
        return name + " (" + country + ")";
    }
}
```

The program prints

```
CH -> [Xue (CH), Bao (CH)]
UK -> [John (UK), Kate (UK)]
PL -> [Janek (PL), Kasia (PL)]
US -> [Mary (US), Cindy (US)]
```

The example below demonstrates combining predicates, and also **filter** function:

Listing 60 LDB-Predicates/Predicates.java

```
import java.util.List;
import java.util.function.Predicate;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class Predicates {
    public static void main (String[] args) {
       Predicate<Integer> p1 = e -> e%2 ==  0;
       Predicate<Integer> p = p1
                             .and(e -> e <= 10)
                             .or(e -> e == 19);
       List<Integer> filteredList =
           Stream.of(1,2,3,4,19,22,12)
          .filter(p)
          .collect(Collectors.toList());
       filteredList.stream().forEach(System.out::println);
       // prints 2, 4, 19
    }
}
```

Several intermediate and terminal operations from previous examples are used below:

Listing 61 LDA-Streams/Streams.java

```java
import java.util.Collections;
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.IntStream;
import java.util.stream.Stream;

enum HairColor { BLACK, BROWN, BLOND, RED, WHITE };
enum EyeColor  { AMBER, BLUE, BROWN, GRAY, GREEN, HAZEL };
enum Sex       { WOMAN, MAN };

public class Streams {
    public static void main(String... aargs) {
        System.out.println("*Sorting by length of name");
        Stream.of("Alice", "Margot", "Mary", "Sue")
           .sorted((a,b) ->
                    Integer.compare(a.length(),b.length()))
           .forEach(System.out::println);

        System.out.println("*Summing ints...");
        int sum = IntStream.of(3,2,9,12,8,4)
           .filter(n -> n%2 == 0)
           .map(n -> 3*n+1)
           .sorted()
           .peek(n -> System.out.print(n+" "))
           .sum();
        System.out.println("\nSum = " + sum);

        List<Person> listp = Arrays.asList(
            new Person("Ann",Sex.WOMAN,
                       HairColor.BLOND,EyeColor.BLUE),
            new Person("Joe",Sex.MAN,
                       HairColor.BLACK,EyeColor.BROWN),
            new Person("Sue",Sex.WOMAN,
                       HairColor.RED,EyeColor.HAZEL),
            new Person("Ben",Sex.MAN,
                       HairColor.BROWN,EyeColor.GREEN),
            new Person("Bea",Sex.WOMAN,
                       HairColor.WHITE,EyeColor.GRAY)
        );

        System.out.println("*Women's names...");
        String womenNames =
            listp.stream()
                 .filter(p -> p.getSex() == Sex.WOMAN)
                 .map(Person::getName)
                 .collect(Collectors.joining(", "));
        System.out.println("Women: " + womenNames);

        System.out.println("*Counting men...");
        long menCount =
            listp.stream()
                 .filter(p -> p.getSex() == Sex.MAN)
                 .count();
        System.out.println("No. of men: " + menCount);

        System.out.println("*Names staring with 'B'");
        String nameB =
            listp.stream()
                 .filter(p -> p.getName().charAt(0) == 'B')
                 .map(Object::toString)
                 .collect(Collectors.joining("\n"));
        System.out.println(nameB);
    }
}

class Person {
    private final String    name;
    private final Sex       sex;
    private final HairColor hairColor;
    private final EyeColor  eyeColor;
    public Person(String n, Sex g,
                  HairColor hc, EyeColor ec) {
        name      =  n;
        sex       =  g;
        hairColor = hc;
        eyeColor  = ec;
    }
    public String getName() { return name; }
    public Sex    getSex() { return sex; }
    public HairColor getHairColor() { return hairColor; }
    public EyeColor getEyeColor() { return eyeColor; }

    @Override
    public String toString() {
        return (sex == Sex.WOMAN ? "Mrs " : "Mr ") +
            name + " (hair:" + hairColor + ", eyes:" +
            eyeColor + ")";
    }
}
```

The program prints

```
*Sorting by length of name
Sue
Mary
```

```
Alice
Margot
*Summing ints...
7 13 25 37
Sum = 82
*Women's names...
Women: Ann, Sue, Bea
*Counting men...
No. of men: 2
*Names staring with 'B'
Mr Ben (hair:BROWN, eyes:GREEN)
Mrs Bea (hair:WHITE, eyes:GRAY)
```

Next example demonstrates **map**, references to a constructor (also, to a 'constructor' of an array) and the **toArray** terminal operation.

Listing 62 LDH-RefConstr/RefConstr.java

```java
import java.util.Arrays;
import java.util.stream.Stream;

public class RefConstr {
    public static void main (String[] args) {
        String[] names = {"Ada", "Bea", "Sue", "Lea" };
        Person[] persons =
            Stream.of(names)
             .map(Person::new)
             .toArray(Person[]::new); // otherwise Object[]
        System.out.println(Arrays.toString(persons));
    }
}

class Person {
    private String name;
    Person(String n)        { name = n; }
    public String getName() { return name; }
    @Override
    public String toString(){ return "Miss " + name; }
}
```

The program prints

```
[Miss Ada, Miss Bea, Miss Sue, Miss Lea]
```

The final example demonstrates **flatMap**, which 'flattens' a stream of streams into one stream, consisting of all elements from these individual streams:

Listing 63 LDI-FlatMap/Flat.java

```java
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.Comparator;
import java.util.stream.Collectors;
import java.util.stream.Stream;
import static java.nio.charset.StandardCharsets.UTF_8;

public class Flat {
    public static void main(String[] args) {
        String result = null;
        try (Stream<String> stream =
                Files.lines(Paths.get("Flat.java"),UTF_8))
        {
            result =
                stream
               .flatMap(l -> Stream.of(l.split("\\P{L}+")))
               .filter(w -> w.length() > 9)
               .distinct()
               .sorted(Comparator.comparing(
                                String::length).reversed())
               .collect(Collectors.joining(", "));
        } catch(IOException ignore) { }
        System.out.println(result);
    }
}
```

The program prints

```
StandardCharsets, IOException, Comparator, Collectors
```

Section 7

Functional interfaces

Functional interfaces are those which declare *only one abstract* method, although they may contain definitions of default methods (marked with the keyword **default**) and static functions. When defining a functional interface, we should, although it is not strictly required, mark them with the annotation **FunctionalInterface** — the compiler will then check if the class definition actually defines a functional interface; e.g.,

```
@FunctionalInterface
interface Calc {
    double calculate(double d);
}
```

The standard library (in package ***java.util.function***) defines several functional interfaces. The definitions are generic, i.e., they are expressed in terms of type parameters which may correspond to different (object) types. There are also versions with primitive types — **int**, **long**, **double** and **boolean**.

Let us briefly mention functional interfaces from the standard library.

7.1 Consumers

Consumers represent operations which accept (‘consume’) one or two arguments but do not return anything; therefore they are used for their side effects.

Interface **Consumer<T>** declares one abstract method of type **void** taking one argument:

```
void accept(T t)
```

where **T** denotes any *object* type. There are also versions for arguments of primitive types: **int**, **long** or **double**

- **IntConsumer** ⟹ `void accept(int t)`
- **LongConsumer** ⟹ `void accept(long t)`
- **DoubleConsumer** ⟹ `void accept(double t)`

Interface **BiConsumer<T,U>** declares one abstract method of type **void** taking two arguments:

```
void accept(T t, U u)
```

where **T** and **U** denote any *object* types. There are also versions with one of the arguments, the *second*, of a primitive type: **int**, **long** or **double**

- **ObjIntConsumer<T>** ⟹ `void accept(T t, int u)`
- **ObjLongConsumer<T>** ⟹ `void accept(T t, long u)`
- **ObjDoubleConsumer<T>** ⟹ `void accept(T t, double u)`

7.2 Functions

Functions represent operations which accept one or two arguments of some (possibly different) types and return a value of a certain type, which may be different from types of arguments (return types are conventionally denoted by the letter R).

Interface **Function<T,R>** declares one abstract method taking one argument and returning a value

```
    R apply(T t)
```

where **T** and **R** denote any *object* types. There are also versions for arguments of primitive types: **int**, **long** or **double**

- **IntFunction<R>** ⟹ `R apply(int t)`
- **LongFunction<R>** ⟹ `R apply(long t)`
- **DoubleFunction<R>** ⟹ `R apply(double t)`

Other versions take argument of an object type but return values of primitive types: **int**, **long** or **double** (note different names of their abstract methods!)

- **ToIntFunction<T>** ⟹ `int applyAsInt(T t)`
- **ToLongFunction<T>** ⟹ `long applyAsLong(T t)`
- **ToDoubleFunction<T>** ⟹ `double applyAsDouble(T t)`

Finally, there are versions with argument and return value of (different) primitive types (note different names of their abstract methods!)

- **IntToLongFunction** ⟹ `long applyAsLong(int t)`
- **IntToDoubleFunction** ⟹ `double applyAsDouble(int t)`
- **LongToIntFunction** ⟹ `int applyAsInt(long t)`
- **LongToDoubleFunction** ⟹ `double applyAsDouble(long t)`
- **DoubleToIntFunction** ⟹ `int applyAsInt(double t)`
- **DoubleToLongFunction** ⟹ `long applyAsLong(double t)`

The cases when both types are the same will be handled by the interface **Operator**.

Interface **BiFunction<T,U,R>** declares one abstract method taking two arguments and returning a value

```
    R apply(T t, U u)
```

where **T**, **U** and **R** denote any *object* types. There are also versions for return value of primitive type: **int**, **long** or **double** (note different names of their abstract methods!)

- **ToIntBiFunction<T,U>** ⟹ `int applyAsInt(T t, U u)`
- **ToLongBiFunction<T,U>** ⟹ `long applyAsLong(T t, U u)`
- **ToDoubleBiFunction<T,U>** ⟹ `double applyAsDouble(T t, U u)`

7.3 Operators

Operators represent functions, for which the return type and the types of argument(s) are all the same (like for ‘normal’ operators: addition, multiplication, etc.). They fall into two categories: unary operators (with one argument) and binary operators (with two arguments). This interface extends **Function**, so the abstract methods have the same names as the corresponding functions.

Interface **UnaryOperator<T>** represents an operation on a single argument that produces a result of the same type as that of the argument; the abstract method is

```
T apply(T t)
```

where T is any *object* type. There are also versions for primitive types (note different names of their abstract methods!)

- IntUnaryOperator ⟹ `int applyAsInt(int t)`
- LongUnaryOperator ⟹ `long applyAsLong(long t)`
- DoubleUnaryOperator ⟹ `double applyAsDouble(double t)`

Interface BinaryOperator<T> represents an operation with two arguments and returning a result: all of the same type (this interface extends BiFunction). The abstract method is therefore

```
T apply(T t1, T t2)
```

where T is any *object* type. There are versions for primitive types (note different names of their abstract methods!)

- IntBinaryOperator ⟹ `int applyAsInt(int t1, int t2)`
- LongBinaryOperator ⟹ `long applyAsLong(long t1, long t2)`
- DoubleBinaryOperator ⟹ `double applyAsDouble(double t1, double t2)`

7.4 Predicates

Predicates are functions returning a logical value: either **true** or **false**.
Iterface Predicate<T> represents a predicate with one argument

```
boolean test(T t)
```

where T is any *object* type. There are versions for primitive types:

- IntPredicate ⟹ `boolean test(int t)`
- LongPredicate ⟹ `boolean test(long t)`
- DoublePredicate ⟹ `boolean test(double t)`

Interface BiPredicate<T,U> represents a predicate with two arguments and its abstract method is

```
boolean test(T t, U u)
```

7.5 Suppliers

Suppliers represent functions which do not take any argument but return a value.
Interface Supplier<T> declares an abstract method

```
T get()
```

where T is any *object* type. There are versions for primitive types (note different names of their abstract methods!)

- BooleanSupplier ⟹ `boolean getAsBoolean()`
- IntSupplier ⟹ `int getAsInt()`
- LongSupplier ⟹ `long getAsLong()`
- DoubleSupplier ⟹ `double getAsDouble()`

7.6 Example

The static function **mapFilter** in the program below expects a list, a **Predicate** and a **Function**. The predicate selects from a list elements satisfying the predicate, while the function transforms them to another type:

Listing 64 EMC-FInterfs/FInterfs.java

```java
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.function.Function;
import java.util.function.Predicate;
public class FInterfs {
    public static void main(String[] args) {
        List<String> ls = Arrays.asList(
            "Jane", "Sue", "Alice", "Kim", "Cecilia");
        List<Character> lc = mapFilter(
            ls, s -> s.length() > 3, s -> s.charAt(0));
        System.out.println(lc);
    }
    static <T,R> List<R> mapFilter(
            List<T> list, Predicate<T> p, Function<T,R> f) {
        List<R> n = new ArrayList<>();
        for (T e : list) if(p.test(e)) n.add(f.apply(e));
        return n;
    }
}
```

In the example above the predicate selects only strings longer than three characters, while the function transforms **String** into **Character**. The program prints `[J, A, C]`.

Section 8

Introduction to multithreading

8.1 Processes and threads

Modern computers can run many applications at the same time, simultaneously. For each such application the operating system creates a separate **process** which gets its address space, standard input and output streams (and also the so called standard error stream), and other resources that the process needs. Usually, however, there are much more processes than available physical processors or cores. Therefore, the operating system has to stop (preempt) some processes, storing their current state (contents of registers, stack, etc.) and load another process in their place for some **time slice**. This operation is called **context switching** and is quite expensive. Basically, we cannot know when and how often these switchings will take place.

A **threads** are kind of a subprocesses executed "inside" a process. Each of them has its own stack, but they all share one address space, in particular the heap, and other resources allocated by the operating system to the parent process. Like processes, they can run concurrently, be preempted at unpredictable moments, etc. Each runs the same or another sequence of actions as the others.

Threads running inside the same process share data on the heap: this ensures easy communication between them. On the other hand, it can happen that two or more threads access the same piece of data and one is modifying it. In such situation it is possible that the data will be read by one thread when only 'half-modified' by another thread. Moreover, if one thread assigns a new value to a variable, this modification may be not visible by other threads, because it was only effectuated in a cache or register. Even worse: compiler can reorder instructions executed by one thread as long as this doesn't change the semantics of a sequence of instructions *from the point of view* of this thread. However, if data is shared, it may happen that the order of these instructions *does* matter from the point of view of other threads! We will therefore need certain means to handle all these situations.

Threads are represented by objects of class **Thread**. The class defines method `public void run()` — its implementation determines what the thread will do. The default implementation of **run** does nothing. Starting the thread will invoke this method, exiting it will the stop the execution of the thread — it cannot be restarted (although the object itself still exists).

Threads have a priority, which can be modified by **setPriority** method — threads with higher priority are supposed to be executed in preference to those with lower priority, but implementation of this mechanism depends on the operation system, so it cannot be relied on.

Threads cannot be "killed" — exiting the **run** method is the only way for the thread to stop execution. Each thread has a boolean flag which indicates if it is interrupted — this flag may be set by another thread, but by itself it does *not* interrupt anything: the thread may detect that its interruption flag is set and "commit suicide" by exiting **run** (or it may just ignore the interruption).

Let us now explain

- how to create and launch a new thread;
- how to ensure integrity of data shared by many threads and synchronize actions on this data.

8.2 Creating threads

There are two ways of creating the object of class **Thread** representing a thread:

- Create a class extending **Thread** and override its **run** method, so it does something useful. Then create an object of this class and invoke method **start()** on it.
- Create a class implementing the functional interface **Runnable**. This interface has one method which has to be implemented: `public void run()`. Then create an object of class **Thread** passing to its constructor an object of your class implementing **Runnable**. Call **start** on this object.

At any moment, a created thread can be in one, and only one, of six different states, represented by constants of enumeration **Thread.State**

1. NEW — a thread exists but has not yet started;
2. RUNNABLE — a thread is being executed by the JVM, although it may be waiting for some resources from the operating system (e.g., preempted thread waiting for a processor);
3. BLOCKED — a thread is blocked waiting for a monitor lock (see below) — as soon as it acquires the lock, it will be in state RUNNABLE again;
4. WAITING — a thread is waiting to be "awaken" (see below); this happens after calling, without any timeout specified, **wait** on a monitor lock or static **Thread.join**;
5. TIMED_WAITING — a thread is waiting to be "awaken", but with a specified waiting time; this happens after calling **Thread.sleep** or, with timeout specified, **wait** on a monitor lock or static **Thread.join**;
6. TERMINATED — a thread is "dead"; it has completed its execution (and cannot be restarted).

Now let us consider an example of a "data race" — several threads access the same variable at the same time:

Listing 65 QKC-BadThreads/BadThreads.java

```java
public class BadThreads extends Thread {

    private long number = 0L;

    public static void main(String[] args) {
        new BadThreads().start();
    }

    public BadThreads() {
        final int MAXNUM = 40;
        for (int i = 0; i < MAXNUM; ++i)
            new Thread(new MyRunner(this),""+i).start();
        System.err.println(MAXNUM + " THREADS STARTED");
    }

    public long getNumber() {
        if (number < 1) number = number + 1;
        number = number - 1;
        return number;
    }

    @Override
    public void run() {
        try {
            Thread.sleep(4000);
        } catch (InterruptedException ignored) { }
        System.err.println("Killing program");
        System.exit(0);
    }
}

class MyRunner implements Runnable {
    private final BadThreads bad;

    public MyRunner(BadThreads bad) {
        this.bad = bad;
    }

    @Override
    public void run() {
        String name = Thread.currentThread().getName();
        while (true) {
            long n = bad.getNumber();
            if (n != 0) {
                System.err.println(
                    "n = " + n + " in thread " + name);
                break;
            }
        }
    }
}
```

The output can be something like:

```
n = -1 in thread 1
n = 1 in thread 2
n = 2 in thread 0
n = 1 in thread 3
n = -1 in thread 5
n = -1 in thread 6
n = -1 in thread 4
n = -1 in thread 7
n = -1 in thread 8
n = -1 in thread 9
n = -1 in thread 10
n = -1 in thread 13
n = -1 in thread 11
```

```
n = -1 in thread 14
n = -1 in thread 12
n = -1 in thread 18
n = -1 in thread 15
n = -1 in thread 16
n = -1 in thread 17
n = -1 in thread 20
n = -1 in thread 19
n = -1 in thread 22
n = -1 in thread 23
n = -1 in thread 21
n = -1 in thread 26
n = -1 in thread 25
n = -1 in thread 27
n = -1 in thread 24
n = -1 in thread 28
n = -1 in thread 30
n = -1 in thread 31
n = -1 in thread 29
n = -1 in thread 33
n = -1 in thread 32
n = -1 in thread 34
n = -1 in thread 35
n = -1 in thread 38
40 THREADS STARTED
n = -1 in thread 37
n = -1 in thread 36
n = -1 in thread 39
Killing program
```

All threads stop prematurely! How to avoid simultaneous access to data by many threads?

8.3 Synchronization

All objects in Java have a hidden field (sometimes called a lock) which can be in two states: closed or open. This allows us to use any object as a monitor lock. Let obj be any object. Then we can synchronize a fragment of code on this object like this

```java
synchronized (obj) {
    // code
}
```

If the execution of a thread encounters a block of code synchronized on an object (obj in this case),

- it will be blocked, if obj is “locked” (closed);
- if it is open, the thread locks it and enters the block. When leaving the block, it releases the lock again.

The block of code synchronized on a lock is called **critical section**. There can be many fragments of code (critical sections), perhaps scattered in different places of the

whole program, synchronized on the same object; only one of them can be executed at any instance of time — the one executed by thread which acquired the lock when it was open and closed it. All other threads which encountered a block synchronized on *exactly the same* object will have to wait until the lock is released (their state is BLOCKED). When this happens, only one of them (there is practically no way to predict which one) will acquire the lock and enter the critical section — all other will still be blocked.
Let us illustrate this:

Listing 66 QKE-BetterThreads/BetterThreads.java

```java
public class BetterThreads extends Thread {

    private long number = 0L;

    public static void main(String[] args) {
        new BetterThreads().start();
    }

    public BetterThreads() {
    final int MAXNUM = 40;
        for (int i = 0; i < MAXNUM; ++i)
            new Thread(new MyRunner(this),""+i).start();
        System.out.println(MAXNUM + " THREADS STARTED");
    }

    public long getNumber() {
        if (number < 1) number = number + 1;
        number = number - 1;
        return number;
    }

    @Override
    public void run() {
        try {
            Thread.sleep(4000);
        } catch (InterruptedException ignored) { }
        System.out.println("Killing program");
        System.exit(0);
    }
}

class MyRunner implements Runnable {
    private final BetterThreads better;

    public MyRunner(BetterThreads better) {
        this.better = better;
    }

    @Override
    public void run() {
        String name = Thread.currentThread().getName();
        long n;
        while (true) {
            synchronized(better) {
                n = better.getNumber();
            }
            if (n != 0) {
                System.out.println(
                    "n = " + n + " in thread " + name);
                break;
            }
        }
    }
}
```

It often happens that there are methods of a class that modify fields of an object of this class and the object is accessible by many threads. We can then synchronize whole methods of this class. This is equivalent to synchronizing the whole body of the method on **this**, i.e.,

```
class AClass {
    synchronized void fun(/* ... */) {
        // ...
    }
}
```

is equivalent to

```
class AClass {
    void fun(/* ... */) {
        synchronized (this) }
            // ...
        }
    }
}
```

Only one synchronized method will be executed at any given instance of time, provided they are all called on *exactly the same object*.
Example:

Listing 67 QKA-SimpleThreads/FibThreads.java

```
public class FibThreads {

    private int counter;

    public static void main(String[] args) {
        new FibThreads();
    }

```

```java
    FibThreads() {
        Runnable[] runs =
            {
                new Fibo(46L,this), new Fibo(44L,this),
                new Fibo(46L,this), new Fibo(45L,this),
                new Fibo(45L,this), new Fibo(45L,this),
            };
        counter = runs.length;

        for (Runnable r : runs)
            new Thread(r).start();

        System.out.println("Exiting from \"main\"");
    }

    synchronized void finished(long arg, long res) {
        counter = counter - 1;
        System.out.println("Fib(" + arg + ") = " + res +
            ". Still running: " + counter);
    }
}

class Fibo implements Runnable {

    private final long  arg;
    private final FibThreads parent;

    static long fibon(long n) {
        return (n < 2) ? n : fibon(n-2) + fibon(n-1);
    }

    Fibo(long n, FibThreads w) {
        arg    = n;
        parent = w;
    }

    @Override
    public void run() {
        System.out.println("Fibo(" + arg + ") starting");
        long res = fibon(arg);
        parent.finished(arg,res);
    }
}
```

This is also possible to synchronize static methods: this is equivalent to synchronizing the whole body of the function on the object of class **Class** representing the class — there is only one such object and it can be referenced to as AClass.class, where **AClass** is the name of a class (or by calling **getClass** on any object of this class):

```java
class AClass {
    synchronized static void fun(/* ... */) {
        // ...
    }
}
```

is equivalent to

```
class AClass {
    static void fun(/* ... */) {
        synchronized (AClass.class) }
            // ...
        }
    }
}
```

It is crucial to always remember which object plays the rôle of a lock in a given context. For example in the code below

```
class Number {
    static double d;
    synchronized static void set(double x) { d = x; }
    synchronized double get() { return d; }
}
```

the (non-static) method **get** is synchronized on **this**, while **set** on Number.class, because it is static. These two functions, both having access to the field `d`, *can* be executed simultaneously, contrary to our intentions.

One may encounter a similar situation when dealing with outer and inner classes:

```
class Outer {
    double n;
    synchronized set(int nn) { n = nn; }
    // ...
    class Inner {
        synchronized int get() { return n; }
        // ...
     }
}
```

Here **set** in synchronized of **this** object of the outer class, while **get** on **this** which points to an object of the inner class. To avoid this inconsistency, we could have synchronized **get** on the object pointed to by **this** from the outer class:

```
class Outer {
    double n;
    synchronized set(int nn) { n = nn; }
    // ...
    class Inner {
        int get() {
            synchronized (Outer.this) {
                return n;
            }
        }
        // ...
     }
}
```

8.4 Inter-thread coordination

A thread which encountered a critical section with a closed lock is in the BLOCKED state. It is ready to continue as soon as the lock becomes open. However, sometimes we want a thread to wait for another thread to finish, before it can continue (e.g., because the other thread prepares some data which is needed by the current thread to be able to proceed). This may be achieved by calling

```
    otherThread.join();
```

where otherThread is the reference to a thread; the current thread (executing this statement) will wait until otherThread has finished.

In another scenario, a thread has to wait for some event to take place before continuing, and this event is somehow controlled by another thread. We then have to change its state to WAITING. There are *two* queues associated with an object playing the rôle of a lock:

- those which are BLOCKED on it, i.e., they are ready to continue as soon as the lock is released, and
- those which are WAITING on this lock, i.e., they do nothing until they are 'woken up' by another thread — at this moment they are transferred to the BLOCKED queue and will be able to continue as soon as the lock has been open.

This scenario can be realized by using methods (from class **Object**)

- **wait**;
- **notify**;
- **notifyAll**.

All these methods must be invoked

- on an object which plays the rôle of the lock of some critical sections;
- inside a critical section guarded by this object.

The sequence of actions is as follows (by lock we mean the object on which critical sections involved are synchronized):

- A thread calls **wait** on lock. After that the thread is in state WAITING, the lock is *released* (open) and the thread becomes idle (doesn't do anything). It is crucial that the lock is released, because another thread will have to "wake up" this thread also being in a critical section guarded by the same lock — this other thread would never be able to enter this critical section to do it, if the lock is closed! The waiting thread will not proceed until it is awoken by another thread.
- Another thread can now enter a critical section guarded by lock, do something (like modifying the value of a field of an object) and then notify ("wake up") the waiting thread by calling **notify** on lock. At this moment, the waiting thread is moved from the queue of waiting threads to the queue of blocked threads — it cannot resume its execution immediately because the other thread, the one which called **notify**, was in a critical section, so lock is at this moment closed.

There is also a version of **wait** which takes an additional argument — time to wait. After this time has elapsed, the waiting thread will be woken up even without notification.

The method **notify** wakes up only one thread waiting on lock (if there are many, it is not known which one). To notify *all* threads waiting on a lock, call **notifyAll** (which should be avoided if not necessary, because it is rather expensive).

8.5 Terminating threads

A thread cannot be “killed” by another thread. In order to terminate a thread, one can set its flag interrupted by calling **interrupt** on an object representing the thread to be stopped. If a thread for which this flag has been set is waiting or sleeping, or calls **wait** or **Thread.sleep()**, a checked exception InterruptedException will be thrown; then, inside the catch clause, the thread may decide what to do (it can completely ignore this signal). Threads may also check their interrupted flag by calling **isInterrupted** on the thread executing a given piece of code. For example, one get the reference to the thread executing the current code and check if it has been interrupted by invoking

```
Thread.currentThread().isInterrupted()
```

which returns true or false.

8.6 Examples

Let us consider a couple of examples.

The first will illustrate the way to stop a thread by modifying a boolean variable canRun. Modifying this variable doesn’t need to be synchronized, because operations on four-byte variables of primitive type are atomic (but not on doubles or longs). However, as it is accessed by two different threads, it should be declared as volatile. This means that it should always be read directly from memory and stored in memory: compiler is not allowed to cache it anywhere (in registers or cache memory). Otherwise, modifications made by one thread wouldn’t be necessarily seen by other threads!

Listing 68 QKF-StopThread/StopThread.java

```java
public class StopThread {
      // booleans are written/read atomically,
      // 'volatile' here to avoid caching value
    static volatile boolean canRun = true;

    public static void main (String[] args) {

        Thread runner = new Thread(() -> {
            while(canRun) {
                System.out.println("still running...");

                try {Thread.sleep(750);}
                catch(InterruptedException ignored) { }
            }
            System.out.println("INTERRUPTED!");
        });
        runner.start();

        try {Thread.sleep(5000);}
        catch(InterruptedException ignored) { }

        canRun = false;
    }
}
```

The next example will illustrate interrupting threads by setting their interrupted flag. Note that **interrupt** does *not* interrupt anything; the 'interrupted' thread must detect the interruption itself and decide what to do — here it just returns from **run** and hence becomes TERMINATED.

Listing 69 QJS-RunThreads/RunThreads.java

```java
public class RunThreads {
    public static void main (String[] args) {
         // ShowTime extends Thread
        Thread tTime = new ShowTime();

         // ShowLett implements Runnable
        Thread tLett = new Thread(new ShowLett());

         // object of anonymous class extending Thread
        Thread tNumb = new Thread() {
            @Override
            public void run() {
                int num = 0;
                while (true) {
                    try {
                        Thread.sleep(2000);
                    } catch(InterruptedException exc) {
                        System.out.println(
                                " |\nNumb interrupted.");
                        return;
                    }
                    System.out.printf(" | N %d", ++num);
                }
            }
        };

         // object of anonymous class extending Thread;
         // using a lambda (as Runnable is functional)
        Thread tHebr = new Thread( () -> {
            int lett = 0x5D0-1;
            while (true) {
                try {
                    Thread.sleep(1750);
                } catch(InterruptedException exc) {
                    System.out.println(
                            " |\nHebr interrupted.");
                    return;
                }
                int c =  0x5D0 + (++lett-0x5D0)%27;
                System.out.printf(" | H %c", (char)c);
            }
        });

        tTime.start();
        tLett.start();
        tNumb.start();
        tHebr.start();
        try {
            Thread.sleep(10*1000);

            tTime.interrupt(); Thread.sleep(3*1000);
            tLett.interrupt(); Thread.sleep(4*1000);
            tNumb.interrupt(); Thread.sleep(3* 200);
            tHebr.interrupt(); Thread.sleep(   200);
        } catch (InterruptedException e) {
            System.out.println("Should never happen!!!");
            System.exit(1);
        }
        System.out.println("ALL DONE");
    }
}

class ShowTime extends Thread {
    @Override
    public void run() {
        int time = 0;
        while (true) {
            try {
                Thread.sleep(1500);
            } catch(InterruptedException exc) {
                System.out.println(" |\nTime interrupted.");
                return;
            }
            int min = ++time/60;
            int sec = time%60;
            System.out.printf(" | T %02d:%02d",min,sec);
        }
    }
}

class ShowLett implements Runnable {
    @Override
    public void run() {
        int lett = 'A'-1;
        while (true) {
            try {
                Thread.sleep(1250);
            } catch(InterruptedException exc) {
                System.out.println(" |\nLett interrupted.");
                return;
            }
            int c =  'A' + (++lett-'A')%26;
            System.out.printf(" | L %c", (char)c);
        }
    }
}
```

Let us now consider another example illustrating inter-thread coordination. In class **Texts** there is room for only one text (variable txt); there is also a boolean value newTxT which is by assumption **true** only if a text has been set by author (class **Author**) but not yet read (taken) by the publisher (class **Publisher**). Therefore, if newTxT is **true**, the author cannot set a new value of txt — he waits until the publisher takes the text and notifies him about this. On the other hand, the publisher cannot proceed if newTxt is **false** — then *he* waits until the author sets a new text and wakes him up (by means of **notify**). Note that the condition newTxT is checked in **while** loop, and *not* by an **if**. In this simple program **if** would be sufficient. However, when more threads are active, **while** has to be used. Imagine that a thread has already executed the **if** statement went on hold. Then another thread modifies the condition variable (here, newTxT) and notifies this thread, which is then transferred to the blocked queue. After the lock has been open, the thread resumes execution, but by this time yet another thread could have changed the variable again — this will not be detected, as the **if** statement has already been executed!

Note also that here the variable newTxT doesn't need to be **volatile**: modifications of variables made inside or before entering a critical section *are* visible by the code synchronized on the same lock which enters its critical section later.

Listing 70 QKG-Coord/Coord.java

```java
class Texts {
    private String txt = null;
    private boolean newTxt = false;

      // invoked by Author to set a new text
    synchronized public void setText(String s) {
        while (newTxt) { // not if!!!
            try {
                wait();
            } catch(InterruptedException exc) {}
        }
        txt = s;
        newTxt = true;
        notify(); // invoked on 'this'
    }

      // invoked by Publisher to get a text
    synchronized public String getText() {
        while (!newTxt) { // not if!!!
            try {
                wait(); // invoked on 'this'
            } catch(InterruptedException exc) {}
        }
        newTxt = false;
        notify(); // invoked on 'this'
        return txt;
    }
}

class Publisher extends Thread {
    private Texts txtArea;
    public Publisher(Texts t) {
        txtArea=t;
    }

    public void run() {
        String txt = null;
        while ((txt = txtArea.getText()) != null) {
            System.out.println("-> " + txt);
        }
    }
}

class Author extends Thread {
    private Texts txtArea;
    public Author(Texts t)  {
        txtArea=t;
    }

    public void run() {
        String[] texts = {"Hamlet", "War and Peace",
            "Macbeth", "The Trial", "Crime and Punishment",
            "Madame Bovary", null };
        for (int i=0; i<texts.length; i++) {
            try {
                  // writing a book takes some time...
                sleep((int)(1500 + Math.random()*300));
            } catch(InterruptedException ignored) { }

            txtArea.setText(texts[i]);
        }
    }
}

public class Coord {
    public static void main(String[] args) {
        Texts t   = new Texts();
        Thread t1 = new Author(t);
        Thread t2 = new Publisher(t);
        t1.start();
        t2.start();
    }
}
```

The next example illustrates stopping and resuming threads. Variables stopped and suspended should be **volatile**, because their values are modified when executing a code which is *not* in a critical section guarded by the lock, so without **volatile** there would be no guarantee that inside a critical section a new, modified value, is visible. Again, it is very important to check the values of these 'condition' variables in a **while** loop, and *not* just by an **if**. Suppose a thread is waiting on suspended (suspended is **true**)

while (suspended) wait();

and another thread sets suspended to **false** and notifies this thread. This thread cannot resume execution immediately — it is now blocked on the lock. When the lock is released, in unforeseeable future, and the thread can finally resume its execution, it may happen that suspended is again **true**, because it has been modified again by another thread! If we check the condition in a loop, it *will* be checked again, while with **if** there would no additional checking and the thread would proceed, although suspended would now be **true**!

Listing 71 QKH-SuspResum/SuspResum.java

```java
import javax.swing.JOptionPane;

class MyThread extends Thread {
    volatile boolean stopped   = false;
    volatile boolean suspended = false;

    public void run() {
        int num = 0;
        while(!stopped) {
            try {
                synchronized(this) {
                    while (suspended) wait();
                }
            } catch (InterruptedException exc) {
                System.out.println(
                        "Interrupted on wait");
            }
            if (suspended) System.out.println(
                               "Still suspended");
            else           System.out.println(++num);
        }
    }

    public void stopThread()    { stopped = true;   }
    public void suspendThread() { suspended = true; }
    public boolean isSusp()     { return suspended; }
    public boolean isStop()     { return stopped;   }

    public void resumeThread() {
        suspended = false;
        synchronized(this) {
            notify();
        }
    }
}

public class SuspResum {
    public static void main(String args[]) {
        String msg = "I = interrupt\n" +
                     "E = end\n" +
                     "S = suspend\n" +
                     "R = resume\n" +
                     "N = new start";
        MyThread t = new MyThread();
        t.start();
        while (true) {
            String cmd = JOptionPane.showInputDialog(msg);
            if (cmd == null) break;
            if (cmd.trim().length() == 0) continue;
            char c = Character.toUpperCase(cmd.charAt(0));
            switch (c) {
                case 'I' : t.interrupt();     break;
                case 'E' : t.stopThread();    break;
                case 'S' : t.suspendThread(); break;
                case 'R' : t.resumeThread();  break;
                case 'N' :
                    if (t.isAlive())
                        JOptionPane.showMessageDialog(
                                null, "Thread alive!!!");
                    else {
                        t = new MyThread();
                        t.start();
                    }
                    break;
                default  : break;
            }
            JOptionPane.showMessageDialog(null,
                    "Command " + cmd + " executed.\n" +
                    "Thread alive? " +
                    (t.isAlive() ? "Y\n" : "N\n") +
                    "Thread interrupted? " +
                    (t.isInterrupted() ? "Y\n" : "N\n") +
                    "Thread suspended? " +
                    (t.isSusp() ? "Y\n" : "N\n") +
                    "Thread stopped? " +
                    (t.isStop() ? "Y\n" : "N")
            );
        }
        System.exit(0);
    }
}
```

The Java standard library provides a multitude of classes which make multi-threaded programming much easier and less error prone. As an example, in the program below, we use a **BlockingQueue** which represents a queue automatically guarded against simultaneous accesses. To its constructor, we pass the required capacity of the queue. The mechanism is a possible solution of the classic *producer-consumer* problem. Threads representing producers add new elements by invoking **put**: if there is no room for a new element because the queue is full (number of its elements reached the capacity), they will be blocked and will have to wait until a consumer has popped at least one element. On the other hand, a consumer thread invoking **take** is blocked when there are no elements in the queue available, and will wait until a producer has supplied a new element. What is important is the fact that synchronization of **put** and **take** operations will be taken care of by the library — we don't have to worry about it.

Listing 72 QKI-BlockQ/BlockQ.java

```java
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;

public class BlockQ {
    public static void main(String[] args) {
        BlockingQueue<Integer> queue =
                new ArrayBlockingQueue<>(10);
        Cons c = new Cons(queue,0);
        c.start();
        Prod p1= new Prod(queue,10);
        Prod p2= new Prod(queue,20);
        p1.start();
        p2.start();
    }

    static class Prod extends Thread {
        private final BlockingQueue<Integer> queue;
        private final int low;
        Prod(BlockingQueue<Integer> queue, int low) {
            this.queue = queue;
            this.low   = low;
        }

        @Override
        public void run() {
            for (int i = 0; i < 10; ++i) {
                try {
                    int d = low + (int)(10*Math.random());
                    System.err.println("Try to put " + d);
                    queue.put(d);
                    System.err.println("       put " + d);
                    sleep(500 + (int)(300*Math.random()));
                } catch(InterruptedException ignore) { }
            }
            try {
                  // poison pill
                queue.put(-1);
            } catch(InterruptedException ignore) { }
        }
    }

    static class Cons extends Thread {
        private final BlockingQueue<Integer> queue;
        private final int poison;
        Cons(BlockingQueue<Integer> queue, int poison) {
            this.queue  = queue;
            this.poison = poison;
        }

        @Override
        public void run() {
            int pills = 0;
            while (pills < 2) { // as there are 2 producers
                try {
                    System.err.println("taking");
                    int d = queue.take();
                    System.err.println(" taken " + d);
                    if (d < poison) {
                        System.out.println(
                                    "Poison pill received");
                        ++pills;
                    }
                    sleep(700 + (int)(300*Math.random()));
                } catch(InterruptedException ignore) { }
            }
        }
    }
}
```

Another useful tool provided by the library is the Timer class. It allows to run a task (represented by an object of a class extending TimerTask with its method **run** overridden) repeatedly: we can set a frequency of running the task (or rather a period) and a delay before running it for the first time. This is illustrated in the program below. First, we create a timer which after 20 seconds will run once the **run** method on an object of an anonymous class extending TimerTask (what will stop the program). In a JOptionPane window, we display a mathematical puzzle and, using a Timer, run repeatedly a task displaying a message prompting the user for an answer and printing the time he/she has already spent on solving this puzzle.

Listing 73 QKN-TimerExample/TimerExample.java

```java
import javax.swing.JOptionPane;
import java.util.Random;
import java.util.Timer;
import java.util.TimerTask;

public class TimerExample {
    static int total   = 0,
               correct = 0;

    public static void main(String[] args) {
          // will be run once only, 20 seconds from now
        new Timer().schedule(new TimerTask() {
                public void run() {
                    System.out.println(correct + "/" +
                            total + " correct answers.");
                    System.exit(0);
                }
            }, 20*1000);
        Random rand = new Random();
        while (true) {
            int a = rand.nextInt(10) + 1;
            int b = rand.nextInt(10) + 1;
            String oper = a + " x " + b + " is?";
            int expected = a * b;
            Timer timer = new Timer();
              // 1 second delay and then every 2 seconds
            timer.schedule(new Prompt(a,b),1000,2000);
            String s = JOptionPane.showInputDialog(
                    null,oper,"Higher math drill",
                    JOptionPane.QUESTION_MESSAGE);
            if (s == null) System.exit(1);
            int ans = 0;
            try {
                ans = Integer.parseInt(s);
            } catch(NumberFormatException e) {
                timer.cancel();
                continue;
            }
            ++total;
            if (ans == expected) {
                ++correct;
                System.out.println("OK");
            }
            else
                System.out.println("Wrong!!!");
            timer.cancel();
        }
    }
}

class Prompt extends TimerTask {
    private String oper;
    long start = System.currentTimeMillis();

    public Prompt(int a, int b) {
        oper = a + " x " + b + " is so easy... ";
    }

    @Override
    public void run() {
        long time = System.currentTimeMillis() - start;
        System.out.println(oper +
            "you've been thinking for " +
            (System.currentTimeMillis()-start) + " ms");
    }
}
```

The last examples illustrate **Executors**: these are objects that accept tasks (in the form of object implementing **Runnable** or **Callable**) and then create threads and start them according to a specified policy. In the example below, the executor can accept any number of tasks, but will never execute more than four of them simultaneously:

Listing 74 QKJ-ThreadPool/ThreadPool.java

```java
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;

public class ThreadPool extends Thread {

    ExecutorService pool = null;

    public static void main(String[] args) {
        new ThreadPool().start();
    }

    ThreadPool() {
        Runnable[] runs =
            {
                new Fibo(40L), new Fibo(46L),
                new Fibo(41L), new Fibo(45L),
                new Fibo(42L), new Fibo(44L),
                new Fibo(43L), new Fibo(43L),
                new Fibo(47L), new Fibo(48L),
                new Fibo(44L), new Fibo(42L),
                new Fibo(45L), new Fibo(41L),
                new Fibo(36L), new Fibo(40L),
            };
```

```
          // pool for 3 concurrent threads
        pool = Executors.newFixedThreadPool(2);

          // submitting 16 threads...
        for (Runnable r : runs)
            pool.execute(r);

          // no other threads will be added
        pool.shutdown();
        System.err.println("Shutdown executed");
    }

    public void run() {
        while (!pool.isTerminated()) {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException ignored) { }
            System.err.println("Still running...");
        }
        System.err.println("All done");
    }
}

class Fibo implements Runnable {

    private final long arg;

    Fibo(long n) {
        arg = n;
    }

    static long fibon(long n) {
        return (n < 2) ? n : fibon(n-2) + fibon(n-1);
    }

    @Override
    public void run() {
        System.err.println("Fibo(" + arg + ") starts");
        long res = fibon(arg);
        System.err.println("Fibo(" + arg + ") completed " +
                           "with res = " + res);
    }
}
```

Executors accept also tasks defined by object of classes implementing the functional Callable interface

```
interface Callable<V> {
    V call() throws Exception;
}
```

Unlike the **run** in a **Runnable**, the **call** method in **Callable** is allowed to throw an exception and, if there was no exception, returns a value. We can pass a callable to an executor by calling **submit** method which returns an object of type **Future<V>**. It represents the future result of the task, which we can get by calling the (blocking) method **get**. If the task has completed successfully, we will get the result. If an exception was thrown inside the **call** method, it will be caught, stored in the future object and rethrown when we call **get** in the form of an **EcecutionException** object (from which we can extract information about the original exception). There are also other useful methods of **Future** class, which allow to check without blocking if the task has been completed or to make an attempt to cancel the task.

In the example below, we use the **invokeAll** method of executors. It takes a collection of tasks, returns a list of **Future**s and blocks until all tasks are completed. We can then get the results from these **Future**s:

Listing 75 QLG-Futures/FuturesExample.java

```java
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Future;

class SingleTask implements Callable<Integer> {
    Integer num;
    public SingleTask(int n) {
        num = n;
    }
      // do NOT handle exceptions here!
    public Integer call() throws Exception {
        Thread.sleep(3000);
        if (num%3 == 0) throw new NumberFormatException();
        return num;
    }
}

public class FuturesExample {

    public static int sum(ExecutorService exec,
                          List<Callable<Integer>> tasks) {
        List<Future<Integer>> results = null;
        try {
            System.out.println("...invoking all");
            results = exec.invokeAll(tasks);
        } catch(InterruptedException e) {
            System.out.println("invokeAll failed");
            System.exit(1);
        }
        exec.shutdown(); // will not wait for other tasks
        System.out.println("Seem like done");

        int sum = 0;
        for (Future<Integer> r : results) {
            try {
                  // get gets the result or rethrows
                  // exception thrown in the call
                int nextint = r.get();
                System.out.println(nextint + " added");
                sum += nextint;
            }
            catch(InterruptedException e) {
                System.err.println("Should not happen");
                System.exit(1);
            }
            catch(ExecutionException e) {
                System.err.println("** ExecutionException " +
                        "caused by " + e.getCause());
                System.err.println("** No value to add, " +
                        "but continuing... ");
            }
        }
        return sum;
    }

    public static void main(String[] args) {
        List<Callable<Integer>> taskList =
            new ArrayList<Callable<Integer>>();
        ExecutorService exec =
                        Executors.newFixedThreadPool(10);
        for (int i=1; i <=5; i++) {
            Callable<Integer> task = new SingleTask(i);
            taskList.add(task);
        }
        int result = sum(exec, taskList);
        System.out.println("Result: " + result);
    }
}
```

Finally, there is the **FutureTask** class, also representing a task — we pass a task to the constructor in the form of a **Callable** or **Runnable** object. The class has a useful protected method **done** which we can override and which will be called automatically when the task completes — either normally or by throwing an exception. Let us see an example:

Listing 76 QLF-Tasks/Tasks.java

```
import java.util.concurrent.Callable;
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.FutureTask;
import java.util.concurrent.TimeUnit;

public class Tasks {
    public static void main(String[] args) {
        FutureTask<Long>[] fs = new MyFutureTask[]{
                new MyFutureTask(new CallableFibo(43)),
                new MyFutureTask(new CallableFibo(45)),
                new MyFutureTask(new CallableFibo(-2)),
                new MyFutureTask(new CallableFibo(47))
            };
        ExecutorService ex =
                Executors.newFixedThreadPool(2);
        for (FutureTask<Long> t : fs) ex.submit(t);
        ex.shutdown();
        try {
            boolean term =
                ex.awaitTermination(10,TimeUnit.SECONDS);
            if (term)
                System.err.println( "All tasks completed");
            else
                System.err.println(
                    "Timeout: some tasks still running");
        } catch(InterruptedException e) {
            System.err.println("Main thread interrupted");
        }
    }
}

class MyFutureTask extends FutureTask<Long> {
    public MyFutureTask(Callable<Long> c) {
        super(c);
    }
    public void done() {
        String mes = "DONE. ";
        if (isCancelled()) mes += "Cancelled.";
        else
            try {
                mes += ("Result OK: " + get());
            } catch(Exception e) {
                mes += ("Exception: " + e.getCause().toString());
            }
        System.err.println(mes);
    }
};

class CallableFibo implements Callable<Long> {
    long arg;
    public CallableFibo(long arg) {
        this.arg = arg;
    }
    public Long call() throws Exception {
        return fibo(arg);
    }
    private long fibo(long n) {
        if (n < 0)
            throw new IllegalArgumentException("From fibo");
        if (n <= 1) return n;
        return fibo(n-1)+fibo(n-2);
    }
}
```

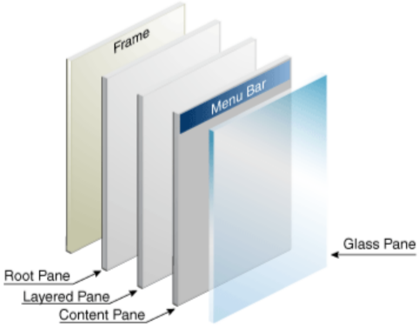
Section 9

GUI - introduction

9.1 Components and containers

Java provides relatively simple tools that can be used to build graphical user interfaces in a way independent of the user's platform. They are collected in two main packages, ***javax.swing*** and ***java.awt*** and their subpackages. Classes defined in these packages describe graphical components (the so called *widgets*, as, for example, windows, buttons, lists, menus, tables) and ways of interactions between the GUI and the user, e.g., by means of reacting to mouse movements and clicks or pressing keys on the keyboard. Generally, we work with GUI components according to the following rules:

- Graphical components are created, like any other Java objects, by using **new** and passing some information to constructors — this information determines properties of the components.
- Components have properties (texts appearing on them, fonts, colors, etc.) — they can be set in a constructor but usually can also be modified and examined dynamically at run time by *setters* (**setXXX**) and *getters* (**getXXX**, **isXXX**), where **XXX** is the name of a property (as color, width, etc.).
- Properties are described by values of primitive types (**int**, **double**) or by objects of classes (as **Font**, **Color**).
- Many components are also **containers**, i.e., we can add to them other components (also other containers).
- Swing windows contain the so called contentPane which is the default container (layer) to which other components may be added. As a matter of fact, it contains many more, although used much less frequently, layers that also can contain components; on top of all layers there is a special layer — 'glass pane' (the picture from Oracle documentation)



- Graphical appearance of the components and their behavior are determined by *layout managers* associated with these components.
- Layout managers are object of special classes and they belong to the properties of components — they may be set for each component separately.
- Applications create one or more windows containing various visual components allowing the user to communicate with the running program.

- Hierarchy of components has as its root a window of the highest level; it is an object of class JFrame, but can be also a JWindow, JApplet or JDialog — these are the so called **heavy-weight components**.

- Communication between the user and the GUI is based on handling **events** (mouse clicks, pressing keys, etc.).

- Events are fetched from the operating system and organized in a queue of events managed by the special thread — the so called **event dispatch thread**).

Historically, the first graphical library in Java was AWT (*Abstract Windowing Toolkit*). Its capabilities were rather limited, many useful graphical components (for example, tables) were missing. Components were 'heavyweight' — implemented in terms of native components provided by a given platform (and, consequently, had different 'look and feel' on various platforms).

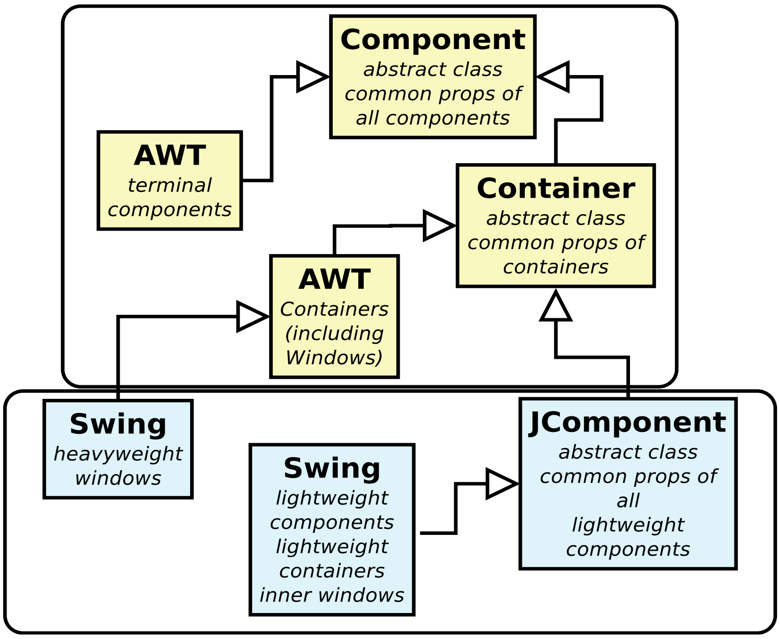
Later, on top of the AWT, a new library was created — the so called **Swing**; it is located in the package *javax.swing* and its subpackages. Swing is much richer than AWT, defines much more components with many useful features. Most of them are **lightweight**, i.e., they are implemented in pure Java without referring to native components of the operating system what implies that they *are* platform independent. There are only a few **heavyweight** components describing main windows of applications, inside which all other components are located. These heavyweight components *are* linked with the native graphical system of the operating system, what is understandable, since ultimately this is the window manager of the platform which is responsible for dealing with windows of all applications running at a given moment. The heavyweight components are JFrame, JApplet, JDialog and JWindow; we will mainly use JFrame.

9.2 Swing components

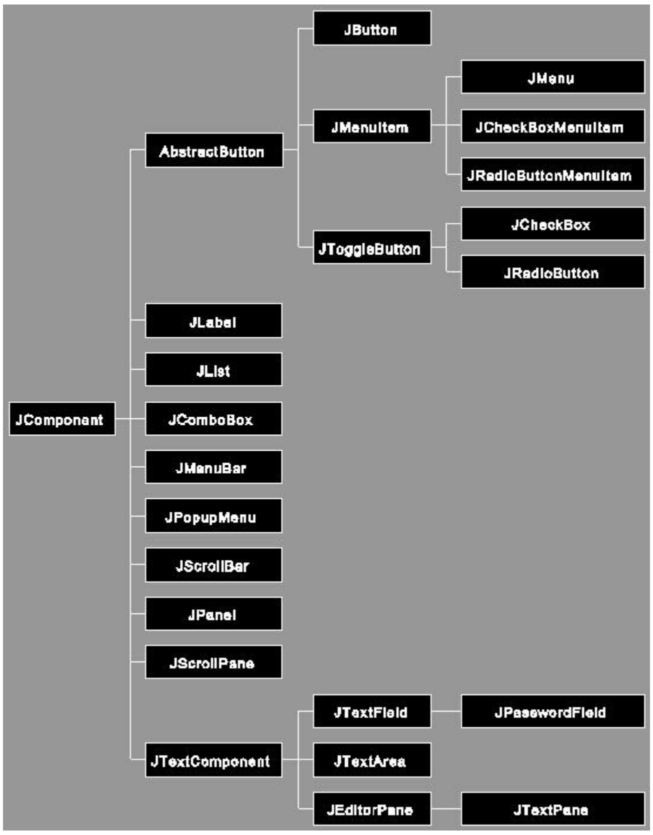
As we said, swing is built on top of the AWT library.

- All components and containers (of both AWT and swing) implement Component from *java.awt*; this interface declares many useful methods for setting and getting properties of all components and containers.
- All containers (both AWT and swing) implement Container from *java.awt*.
- JComponent determines common properties of all lightweight swing components.
- Specific properties and functionality of components are defined in classes of these components.
- Heavyweight swing containers are *not* JComponents — they inherit directly from AWT Container.

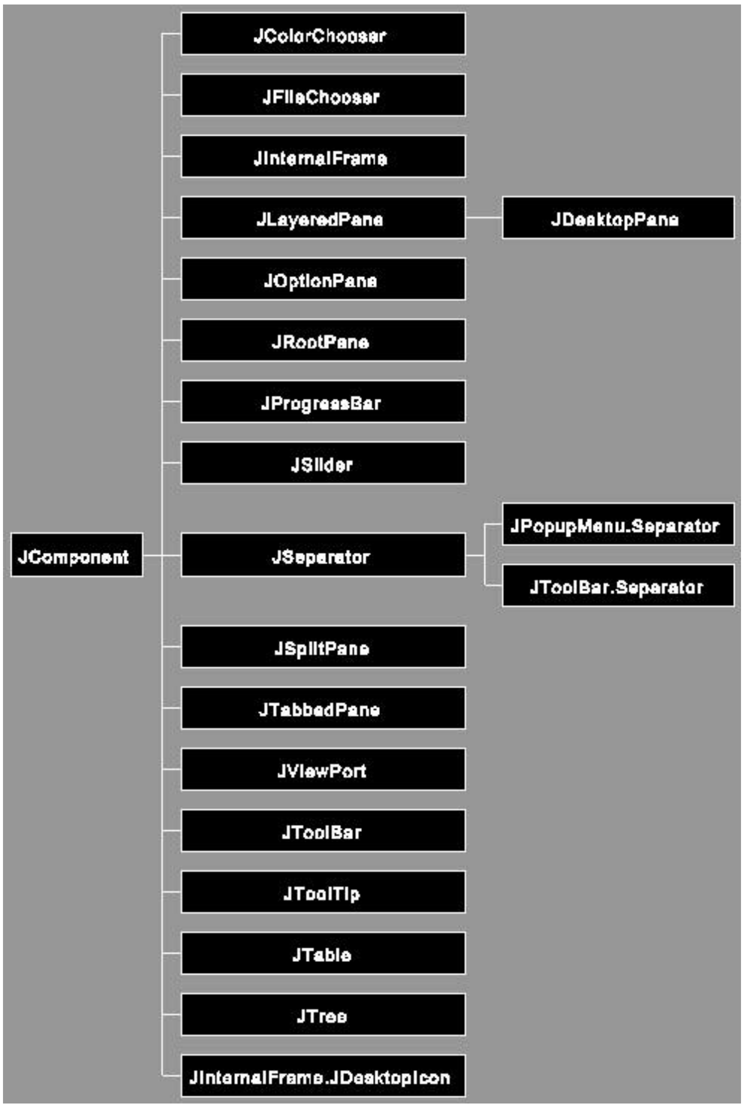
Therefore, the hierarchy looks like this:



The following picture presents **JComponent**s from Swing that extend the corresponding widgets from AWT (from *Magellan Institute Swing Short Course*):



Swing provides much more components; many of them do not have their counterparts in AWT. They are presented in the following figure, also taken from *Magellan Institute Swing Short Course*). Some of them are very elaborated (and often not so easy to use), like **JTable**, **JTree** and some others, but generally it is not difficult to use them, at least on a basic level.



Let us briefly describe swing components and their functionality:

- Buttons: JButton, JToggleButton, JCheckBox, JRadioButton — may have a text and/or an icon with arbitrary positioning, with different text or icon for different states (enabled, disabled), mouse-over effects, may have borders, mnemonics and tips attached, may react to clicks with a mouse or programmatically and so on.
- Labels: JLabel — may have a text and/or an icon with arbitrary positioning, mnemonics, can be linked with another component so clicking its alt-mnemonic transfers the focus to the other component, etc.
- Menus: JMenu, JMenuItem, JCheckBoxMenuItem, JRadioMenuItem — have all properties of buttons. Additionally there are context menus: JPopupMenu.
- Sliders: JSlider — configurable range, description, icons etc.
- Color and file choosers: JColorChooser, JFileChooser — configurable widgets allowing the user to select colors or files (directories); may be embedded in other components.
- One-line editing widgets: JTextField, JPasswordField, JFormattedTextField — entering texts, possible verification; special widget for entering sensitive data (without creating String objects).
- Multi-line text widget: JTextArea, JEditorPane, JTextPane — editing multi-line texts with formatting, embedded graphics etc.

- Lists: **JList** — widget displaying a list of object; fully configurable and dynamic (reflecting modifications of the list at run time).
- Combo boxes: **JComboBox** — similar to **JList** but space-saving.
- Tables: **JTable** — extremely configurable representation of tables; columns of different types, custom rendering of cells and columns, sorting rows etc.
- Trees: **JTree** — configurable representation of data stored in the tree-like form.
- 'Helper' containers: **JPanel**, **JSplitPane**, **JTabbedPane**, **JScrollPane**, — allows the user to group and organize components in various configurable ways.
- Tool bars: **JToolBar** — configurable tool bars for easy launching various actions

Some of them are used in the example below (taken from the Oracle's Swing Tutorial)

Listing 77 MBD-Demo/BasicDnD.java

```java
/*
 * Taken from:
 * https://docs.oracle.com/javase/tutorial/uiswing/examples/
 *          dnd/BasicDnDProject/src/dnd/BasicDnD.java
 * Slightly modified to avoid some warnings
 */
import java.awt.*;
import java.awt.event.*;
import java.awt.datatransfer.*;
import java.text.*;
import java.util.*;
import javax.swing.*;
import javax.swing.table.*;
import javax.swing.text.*;
import javax.swing.tree.*;

public class BasicDnD extends JPanel
        implements ActionListener {
    private static JFrame frame;
    private JTextArea textArea;
    private JTextField textField;
    private JList<String> list;
    private JTable table;
    private JTree tree;
    private JColorChooser colorChooser;
    private JCheckBox toggleDnD;

    public BasicDnD() {
        super(new BorderLayout());
        JPanel leftPanel = createVerticalBoxPanel();
        JPanel rightPanel = createVerticalBoxPanel();

        //Create a table model.
        DefaultTableModel tm = new DefaultTableModel();
        tm.addColumn("Column 0");
        tm.addColumn("Column 1");
        tm.addColumn("Column 2");
        tm.addColumn("Column 3");
        tm.addRow(new String[]{"Table 00", "Table 01",
                               "Table 02", "Table 03"});
        tm.addRow(new String[]{"Table 10", "Table 11",
                               "Table 12", "Table 13"});
        tm.addRow(new String[]{"Table 20", "Table 21",
                               "Table 22", "Table 23"});
        tm.addRow(new String[]{"Table 30", "Table 31",
                               "Table 32", "Table 33"});

        //LEFT COLUMN
        //Use the table model to create a table.
        table = new JTable(tm);
        leftPanel.add(
                createPanelForComponent(table, "JTable"));

        //Create a color chooser.
        colorChooser = new JColorChooser();
        leftPanel.add(createPanelForComponent(
                    colorChooser, "JColorChooser"));

        //RIGHT COLUMN
        //Create a textfield.
        textField = new JTextField(30);
        textField.setText("Favorite foods:" +
                          "\nPizza, Moussaka, Pot roast");
        rightPanel.add(createPanelForComponent(
                           textField, "JTextField"));

        //Create a scrolled text area.
        textArea = new JTextArea(5, 30);
        textArea.setText("Favorite shows:" +
                         "\nBuffy, Alias, Angel");
        JScrollPane scrollPane = new JScrollPane(textArea);
        rightPanel.add(createPanelForComponent(
                           scrollPane, "JTextArea"));

        //Create a list model and a list.
        DefaultListModel<String> listModel =
                                new DefaultListModel<>();
        listModel.addElement("Martha Washington");
        listModel.addElement("Abigail Adams");
        listModel.addElement("Martha Randolph");
        listModel.addElement("Dolley Madison");
        listModel.addElement("Elizabeth Monroe");
        listModel.addElement("Louisa Adams");
        listModel.addElement("Emily Donelson");
        list = new JList<>(listModel);
        list.setVisibleRowCount(-1);
        list.getSelectionModel().setSelectionMode(
                ListSelectionModel.
                MULTIPLE_INTERVAL_SELECTION);

        list.setTransferHandler(new TransferHandler() {
            public boolean canImport(
                TransferHandler.TransferSupport info) {
                // we only import Strings
                if (!info.isDataFlavorSupported(
                        DataFlavor.stringFlavor)) {
                    return false;
                }

                JList.DropLocation dl = (JList.DropLocation)
                                      info.getDropLocation();
                if (dl.getIndex() == -1) {
                    return false;
                }
                return true;
            }

            public boolean importData(
                    TransferHandler.TransferSupport info) {
                if (!info.isDrop()) {
                    return false;
                }

                // Check for String flavor
                if (!info.isDataFlavorSupported(
                            DataFlavor.stringFlavor)) {
                    displayDropLocation(
                            "List doesn't accept a " +
                            "drop of this type.");
                    return false;
                }
                JList.DropLocation dl = (JList.DropLocation)
                                     info.getDropLocation();
                DefaultListModel<String> listModel =
                    (DefaultListModel<String>)
                    list.getModel();
                int ind = dl.getIndex();
                boolean insert = dl.isInsert();
                // Get the current string under the drop.
                String value = listModel.getElementAt(ind);

                // Get the string that is being dropped.
                Transferable t = info.getTransferable();
                String data;
                try {
                    data = (String)t.getTransferData(
                            DataFlavor.stringFlavor);
                }
                catch (Exception e) { return false; }

                // Display a dialog with drop information.
                String dropValue = "\"" +
                                   data + "\" dropped ";
                if (dl.isInsert()) {
                    if (dl.getIndex() == 0) {
                        displayDropLocation(dropValue +
                                "at beginning of list");
                    } else if (dl.getIndex() >=
                               list.getModel().getSize()) {
                        displayDropLocation(
                                dropValue +
                                "at end of list");
                    } else {
                        String value1 =
                            list.getModel()
                           .getElementAt(dl.getIndex() - 1);
                        String value2 = list.getModel()
                                       .getElementAt(dl
                                       .getIndex());
                        displayDropLocation(dropValue +
                                "between \"" + value1 +
                                "\" and \"" + value2 +
                                "\"");
                    }
                } else {
                    displayDropLocation(dropValue +
                            "on top of " + "\"" +
                            value + "\"");
                }
                        return false;
            }

            public int getSourceActions(JComponent c) {
                return COPY;
            }

            @SuppressWarnings("unchecked")
            protected Transferable createTransferable(
                    JComponent c) {
                JList<String> list = (JList<String>)c;
                Object[] values =
                    list.getSelectedValuesList().toArray();

                StringBuffer buff = new StringBuffer();

                for (int i = 0; i < values.length; i++) {
```

```
                    Object val = values[i];
                    buff.append(val == null
                            ? ""
                            : val.toString());
                    if (i != values.length - 1) {
                        buff.append("\n");
                    }
                }
                return new StringSelection(buff.toString());
            }
        });
        list.setDropMode(DropMode.ON_OR_INSERT);

        JScrollPane listView = new JScrollPane(list);
        listView.setPreferredSize(new Dimension(300, 100));
        rightPanel.add(createPanelForComponent(listView,
                                              "JList"));

        //Create a tree.
        DefaultMutableTreeNode rootNode =
            new DefaultMutableTreeNode("Mia Familia");
        DefaultMutableTreeNode sharon =
            new DefaultMutableTreeNode("Sharon");
        rootNode.add(sharon);
        DefaultMutableTreeNode maya =
            new DefaultMutableTreeNode("Maya");
        sharon.add(maya);
        DefaultMutableTreeNode anya =
            new DefaultMutableTreeNode("Anya");
        sharon.add(anya);
        sharon.add(new DefaultMutableTreeNode("Bongo"));
        maya.add(new DefaultMutableTreeNode("Muffin"));
        anya.add(new DefaultMutableTreeNode("Winky"));
        DefaultTreeModel model =
            new DefaultTreeModel(rootNode);
        tree = new JTree(model);
        tree.getSelectionModel().setSelectionMode
              (TreeSelectionModel
              .DISCONTIGUOUS_TREE_SELECTION);
        JScrollPane treeView = new JScrollPane(tree);
        treeView.setPreferredSize(new Dimension(300, 100));
        rightPanel.add(createPanelForComponent(treeView,
                                              "JTree"));

        //Create the toggle button.
        toggleDnD = new JCheckBox("Turn on Drag and Drop");
        toggleDnD.setActionCommand("toggleDnD");
        toggleDnD.addActionListener(this);

        JSplitPane splitPane = new JSplitPane(
                JSplitPane.HORIZONTAL_SPLIT,
                leftPanel, rightPanel);
        splitPane.setOneTouchExpandable(true);

        add(splitPane, BorderLayout.CENTER);
        add(toggleDnD, BorderLayout.PAGE_END);
        setBorder(BorderFactory.createEmptyBorder(5,5,5,5));
    }

    protected JPanel createVerticalBoxPanel() {
        JPanel p = new JPanel();
        p.setLayout(new BoxLayout(p, BoxLayout.PAGE_AXIS));
        p.setBorder(BorderFactory
                  .createEmptyBorder(5,5,5,5));
        return p;
    }

    public JPanel createPanelForComponent(JComponent comp,
                                          String title) {
        JPanel panel = new JPanel(new BorderLayout());
        panel.add(comp, BorderLayout.CENTER);
        if (title != null) {
            panel.setBorder(
                BorderFactory.createTitledBorder(title));
        }
        return panel;
    }

    private void displayDropLocation(final String string) {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                JOptionPane.showMessageDialog(null, string);
            }
        });
    }

    public void actionPerformed(ActionEvent e) {
        if ("toggleDnD".equals(e.getActionCommand())) {
            boolean toggle = toggleDnD.isSelected();
            textArea.setDragEnabled(toggle);
            textField.setDragEnabled(toggle);
            list.setDragEnabled(toggle);
            table.setDragEnabled(toggle);
            tree.setDragEnabled(toggle);
            colorChooser.setDragEnabled(toggle);
        }
    }

    private static void createAndShowGUI() {
        frame = new JFrame("BasicDnD");
        frame.setDefaultCloseOperation(
                JFrame.EXIT_ON_CLOSE);

        JComponent newContentPane = new BasicDnD();
        newContentPane.setOpaque(true);
        frame.setContentPane(newContentPane);

        frame.pack();
        frame.setVisible(true);
    }

    public static void main(String[] args) {
        javax.swing.SwingUtilities.invokeLater(
            new Runnable() {
                public void run() {
                        UIManager.put("swing.boldMetal",
                                   Boolean.FALSE);
                createAndShowGUI();
            }
        });
    }
}
```

which produces

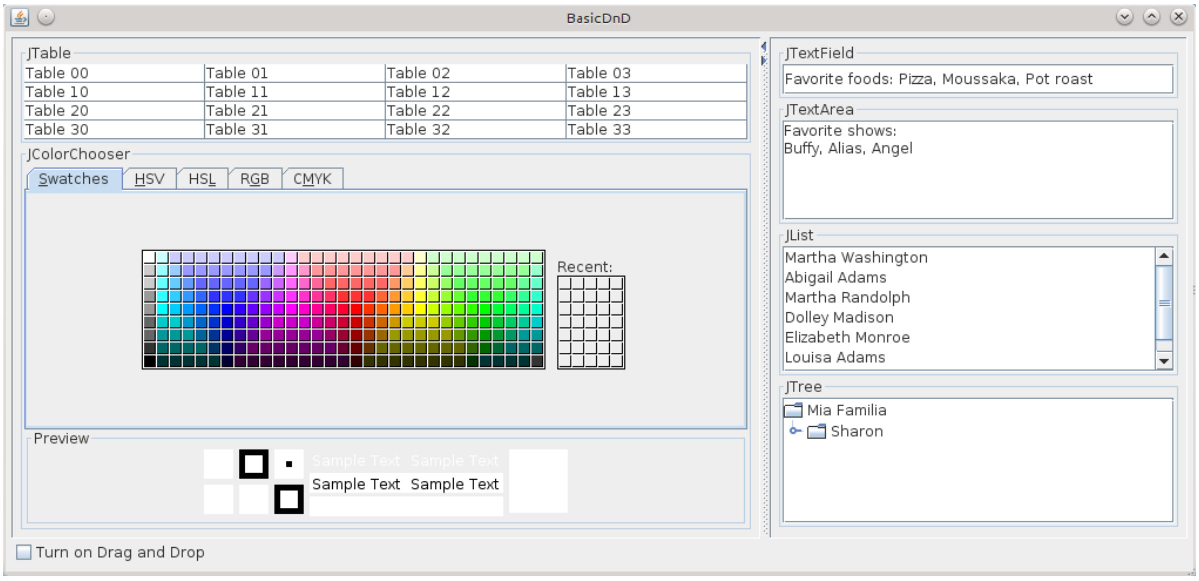


Table 00	Table 01	Table 02	Table 03
Table 10	Table 11	Table 12	Table 13
Table 20	Table 21	Table 22	Table 23
Table 30	Table 31	Table 32	Table 33

Favorite foods: Pizza, Moussaka, Pot roast

Favorite shows:
Buffy, Alias, Angel

Martha Washington
Abigail Adams
Martha Randolph
Dolley Madison
Elizabeth Monroe
Louisa Adams

Mia Familia
Sharon

□ Turn on Drag and Drop

All components are ultimately derived from the abstract class **Component** which defines many methods common for all components. These are mainly getters and setters for properties, such as sizes, colors etc. They are named according to the following convention: for property **prop** the getter is named **getProp** while the setter will be **setProp**. If a property is of logical type (**true** or **false**), then instead of **getProp**, we rather use **isProp**.

Let us mention some of these properties:

- size — determined by an object of type Dimension (from *java.awt*) with fields describing width and height (getWidth, getHeight);
- minimumSize, maximumSize, preferredSize — determined by an object of type Dimension and taken into account by layout managers (not always, though);
- width — width of the component;
- height — height of the component;
- bounds — determined by an object of type Rectangle (from *java.awt*) with fields describing coordinates x and y of the upper-left corner and width and height of the component;
- location — determined by an object of type Point (from *java.awt*) with fields describing coordinates x and y of the upper-left corner of the component;
- alignmentX, alignmentY — determined by a float; indicates alignment along the axes;
- font — determined by an object of type Font from *java.awt* (see below);
- background, foreground — determined by objects of type Color from *java.awt* (see below);
- parent — a Container which contains a given component (read only);
- name — the name of this component; if not set, the system will provide a default one;
- visible — boolean value indicating if the component is visible;
- lightweight — boolean value indicating if this component is lightweight;
- opaque — boolean value indicating if this component is opaque;
- enabled — boolean value indicating if this component can react to events (as, for example, mouse clicks).

One has to remember that all sizes of components are not known until they are 'realized' — this usually happens when methods setSize or pack or setVisible is called on the frame window.

Coordinates of components are alway expressed in the coordinate system where the point $(0, 0)$ corresponds to the upper-left corner; x-coordinate goes from left to right, while y-coordinate goes *downwards* (*sic!*).

Locations and sizes of components inside a container are normally calculated by a layout manager — we can suggest our preferences by setting them 'by hand', but that is only a suggestion... However, we *can* set the size of the main frame window by invoking, e.g., `frame.setSize(200, 200)` on it. The sizes of all components inside it will then be determined by a layout manager. Or, we can pack the main window (`frame.pack()`) and its size will be determined by its contents.

Fonts are specified by objects of type Font (from *java.awt*); its main constructor takes three arguments

```
new Font(String name, int style, int size)
```

where

- name is the font name (case insensitive). However, specifying a concrete font name may be a little bit risky, because we don't know if such a font is available on the user's system. Therefore, we can only specify a generic (logical) name of the font we want — there are five such names: Dialog, DialogInput, Monospaced, SansSerif and Serif. The most appropriate font from those installed on a given system will then be selected.

- style is a static final integer defined in class **Font**: Font.PLAIN, Font.BOLD or Font.ITALIC. They can be "OR'ed", e.g. `Font.BOLD | Font.ITALIC`.
- size is an integer specifying the size (in points, where point is 1/72 of an inch).

Colors are described by objects of class **Color** (from ***java.awt***). The main constructor takes three integers specifying red, green and blue components (or four integers, with transparency, the so called α-channel, added). They all should have values from the range $[0, 255]$. We can create such object like this:

```
new Color(255,33,33)
```

However, the most popular colors are already defined as static fields of class **Color** that are themselves objects of this class: these are BLACK, BLUE, CYAN, DARK_GRAY, GRAY, GREEN, LIGHT_GRAY, MAGENTA, ORANGE, PINK, RED, WHITE, and YELLOW.

Components can react to events like mouse clicks or pressing a key when they have the focus. This feature can be dynamically disabled or enabled (by invoking `comp.setEnabled(false)` or `comp.setEnabled(true)`).

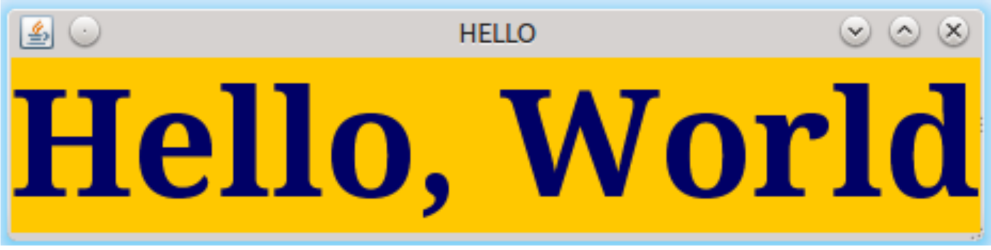
9.3 Swing program

Let us now consider a very simple example:

Listing 78 MBC-HelloG/HelloWorldG.java

```java
import java.awt.Color;
import java.awt.Font;
import javax.swing.JFrame;
import javax.swing.JLabel;

public class HelloWorldG {
    public static void main(String[] args) {
        //
        // Should be on the EDT!
        //
        JFrame fr = new JFrame("HELLO");
        fr.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JLabel label = new JLabel("Hello, World");
          // all properties have reasonable defaults, but...
        label.setFont(new Font("Serif",Font.BOLD,70));
        label.setBackground(Color.ORANGE);
        label.setOpaque(true);
        label.setForeground(new Color(0,0,102));

        fr.add(label); // frame.getContentPane().add(label)

        fr.pack();
          // fr.setSize(600,400);
        fr.setLocationRelativeTo(null);
        fr.setVisible(true);
    }
}
```

which displays


HELLO

Hello, World

Main points to notice here

- fr represents the frame window: it is a heavyweight container of type JFrame into which we will put all components.
- By invoking setDefaultCloseOperation we specify what will happen when the window is closed. This is determined by an integer (it should be an enum...) defined in class JFrame (strictly speaking inherited by implementing the interface WindowConstants)): EXIT_ON_CLOSE, DISPOSE_ON_CLOSE, DO_NOTHING_ON_CLOSE or HIDE_ON_CLOSE.
- Components may be created in any order; when created, they are just objects in memory, not related to other objects.
- Each existing object can be configured separately (color, font, etc.).
- To put one component into another, we use add on the parent component, passing a child component as the argument.
- Exact sizes of all components are unspecified until pack (or setSize) is invoked on the main window. At this moment, all sizes are calculated and components are arranged inside the frame window (normally this is performed by a layout manager).
- In order to make the window with its contents visible on the screen, we have to call setVisible on it.

9.4 Delegation Event model

Normally, we want our GUI to be responsive — we would like something to happen, for example, when the user clicks the mouse on a button visible on the screen. For this to be possible, any Swing application has to 'listen' to events generated by the system: mouse clicks or moves, key presses, etc. These events are provided by the operating system and can be intercepted by the JVM, which wraps them into objects of a type derived from EventObject and enqueues them on a special FIFO queue for the **Event-Dispatch Thread** (EDT) to process. When the time comes, the EDT pops them from the queue and passes them to listeners.
Events are handled by invoking call-back methods on objects which have been registered as **listeners** of events originating from a given source (e.g., a graphical component, like a button) and of a specified type.

In order to handle events, we need

- a source of events; this can be a graphical component of our GUI or an object representing some data structure. The source holds a collection of its ‘listeners’.
- a way to add (and remove) listeners of events of a specified type and taking place on a given source;
- listeners — objects of classes implementing an appropriate interface and therefore providing definitions of its abstract methods; these methods will be invoked by source objects on listeners as a reaction to an event.

The scheme as described above is called the **delegation event model**.

To one source, we can attach many listeners, and the other way around: one listener can be attached to many sources. In order to delegate a listener as a handler of events, we usually invoke a special method

```
source.addXXXListener(listener);
```

where `XXX` specifies the type of events we are interested in; it can be, for example, `Action`, `Mouse`, `MouseMotion`, `Key` etc. The reference `source` refers to an object that has the ability to fire events of the specified type. As the argument, we pass an object which can handle events of the given type: its class has to implement a special interface which declares (as abstract methods) actions that are to be executed after an event of the given type occurred on the given source. Event-handling methods are **public void** and they accept one argument: an object which carries information on the event, as, for example, the source of the event, time of occurrence, and other properties depending on the type of this particular event.

To avoid conflicts, the EDT should be the *only* thread which directly interacts (and therefore can modify) the GUI: it is very important to remember that after the EDT has been started

> (almost) all operations modifying the GUI should be executed on the event dispatch thread.

As the standard doesn’t state clearly when exactly the EDT is launched, it is recommended to perform *all* operations on Swing components, even before displaying them on the screen, on the EDT. We can do it by invoking the static method **invokeLater(Runnable)** from class **SwingUtilities** (or **EventQueue**). The method takes a **Runnable** and the operations we want to perform have to be contained in its **run** method: we can do it by passing an object of our own class implementing **Runnable**, or an object of an anonymous class, or a lambda, because **Runnable**, having only one abstract method **run**, is a functional interface:

```
SwingUtilities.invokeLater(new MyRunnable(...));

SwingUtilities.invokeLater(new Runnable() {
    @Override
    public void run() {
        // ...
    }
});

SwingUtilities.invokeLater( () -> {
    // ...
});
```

Our **Runnable** will be wrapped in an object representing an event and inserted into the event queue, from where it will be popped and executed *on the EDT*.

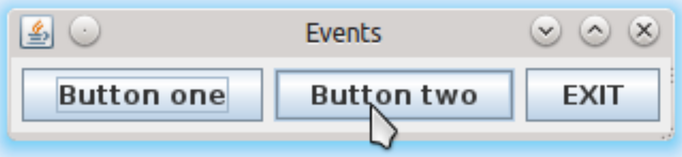
Let us consider an example:

Listing 79 MBH-Events/Events.java

```java
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.SwingUtilities;

public class Events {
    public static void main(String[] args) {
        SwingUtilities.invokeLater( () -> createGUI() );
    }
    private static void createGUI() {
        JFrame f = new JFrame("Events");
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

          // these will be sources
        JButton b1 = new JButton("Button one");
        JButton b2 = new JButton("Button two");
        JButton ex = new JButton("EXIT");

          // listener (as a lambda) implementing
          // public void actionPerformed(ActionEvent)
        ActionListener lis = e -> {
            String s = ((JButton)e.getSource()).getText();
            System.out.println(s + " clicked");
        };

          // registering one listener with two buttons
        b1.addActionListener(lis);
        b2.addActionListener(lis);
          // registering listener of 'exit' button
        ex.addActionListener(e -> System.exit(0));

        JPanel p = new JPanel();
          // adding buttons to the panel
        p.add(b1);
        p.add(b2);
        p.add(ex);
          // adding panel to the frame
        f.add(p);
          // now all sizes will be calculated;
          // do not add anything after packing!
        f.pack();
          // the window will be centerd on the screen
        f.setLocationRelativeTo(null);
```

```
          // show the window
        f.setVisible(true);
    }
}
```

which displays a simple GUI



reacting to clicks on the buttons.

9.5 Layouts

Each container has, associated with it, a layout manager which is responsible for arranging components contained in the container — initially and also after resizing it. The managers are objects of classes implementing the interface LayoutManager from *java.awt*. They can (but do not have to) take into account sizes suggested by the user who may invoke, on any component, methods **setPreferredSize**, **setMaximumSize** and **setMinimumSize** passing an object of class Dimension (with only two fields: width and height).

It can happen that sizes and locations of components in a given container change or a new child components is added: in such situation calling **revalidate** may help (if not, try also **repaint**). Also, calling **pack** on the parent widow will recalculate all sizes and locations of the child components.

On any container, we can invoke **setLayout** passing an object representing a layout manager. There are five main layout managers that we can use. In fact there are more, but others are harder to use; they are extensively used by graphical tools which automate the process of building the GUI. However, the five basic managers that we will present are quite sufficient in vast majority of cases and are easy to use in programs written ‘by hand’.

9.5.1 FlowLayout

Components added to a container (by invoking **add(component)** on it) will be arranged in one row, from left to right; if there is no room for a component in the current row, the second row will be added, and so on. The FlowLayout class has three constructors:

- `FlowLayout(int align, int hgap, int vgap)` — the first argument determines the alignment; it is an integer constant from class FlowLayout: LEFT, CENTER (the default) or RIGHT. The other two integers specify horizontal and vertical (if there is more than one row) gaps between components, and also gaps between the components and the borders;

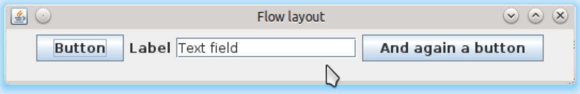
- FlowLayout(int align) — equivalent to FlowLayout(align, 5, 5);
- FlowLayout() — equivalent to FlowLayout(CENTER, 5, 5).

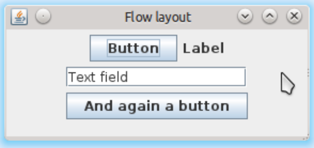
A simple example:

Listing 80 MBI-Flow/FlowEx.java

```java
import java.awt.FlowLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JTextField;
import javax.swing.SwingUtilities;

public class FlowEx extends JFrame {
    public static void main (String[] args) {
        SwingUtilities.invokeLater(() -> new FlowEx());
    }
    FlowEx() {
        super("Flow layout");
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setLayout(new FlowLayout());
        add(new JButton("Button"));
        add(new JLabel("Label"));
        add(new JTextField("Text field", 15));
        add(new JButton("And again a button"));
        pack();
        setLocationRelativeTo(null);
        setVisible(true);
    }
}
```

displaying, depending on the width of the window, the components in one or more rows





9.5.2 GridLayout

In this layout, components will be added to a grid of rectangular cells of the same size (in the order row by row, in each row from left to right). The class **GridLayout** has three constructors:

- GridLayout(int rows, int cols, int hgap, int vgap) — the grid will have rows rows and cols columns with the specified horizontal and vertical gaps in-between. One (but not both), of rows and cols can be zero, which means 'as many as needed';

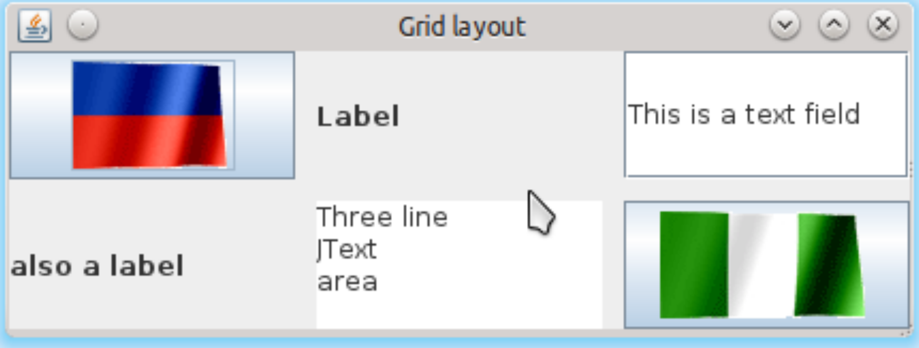
- `GridLayout(int rows, int cols)` — equivalent to `GridLayout(rows, cols, 0, 0);`
- `GridLayout()` — equivalent to `GridLayout(1, 0, 0, 0)` — all components will be added to one row (but they will be of equal sizes).

An example:

Listing 81 MBJ-Grid/GridEx.java

```java
import java.awt.GridLayout;
import javax.swing.ImageIcon;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JTextArea;
import javax.swing.JTextField;
import javax.swing.SwingUtilities;

public class GridEx extends JFrame {
    public static void main (String[] args) {
        SwingUtilities.invokeLater(() -> new GridEx());
    }
    GridEx() {
        super("Grid layout");
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setLayout(new GridLayout(2, 3, 10, 10));
        add(new JButton(new ImageIcon("haiti.gif")));
        add(new JLabel("Label"));
        add(new JTextField("This is a text field"));
        add(new JLabel("also a label"));
        add(new JTextArea("Three line\nJText\narea", 3, 10));
        add(new JButton(new ImageIcon("nigeria.gif")));
        pack();
        setLocationRelativeTo(null);
        setVisible(true);
    }
}
```

displays



9.5.3 BorderLayout

A container with this layout is divided into five areas: NORTH, SOUTH, WEST, EAST and CENTER (corresponding to integer constants in the class GridLayout with these names). We add components by calling **add(component,where)**, where where is one of the constants just mentioned, or CENTER if not specified. Each area can hold only one component, but this component may contain inside it other components. The class BorderLayout has only two constructors:

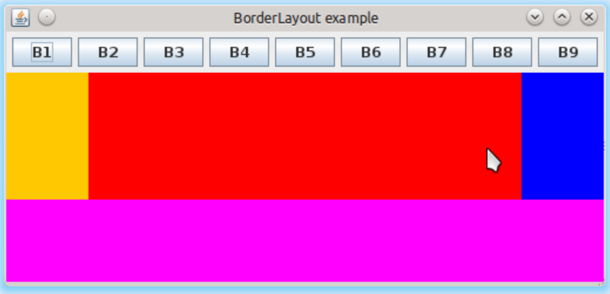
- `BorderLayout(int hgap, int vgap)` — the arguments specify horizontal and vertical gaps between components;
- `BorderLayout()` — equivalent to `BorderLayout(0, 0)`.

When a container with the border layout is resized, the *height* of the NORTH and SOUTH areas are kept constant, while for WEST and EAST their *width* is constant. The CENTER area is scaled in both directions and 'swallows' all empty areas.
This is illustrated in the example below:

Listing 82 MBG-BrdLay/BorderLayoutEx.java

```java
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.FlowLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class BorderLayoutEx extends JFrame {
    public static void main (String[] args) {
        new BorderLayoutEx("BorderLayout example");
    }
    BorderLayoutEx(String title) {
        super(title);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setLayout(new BorderLayout());
        JPanel north = new JPanel();
        north.setLayout(new FlowLayout(FlowLayout.CENTER));
        for (int i = 1; i <= 9; ++i)
            north.add(new JButton("B" + i));
        add(north, BorderLayout.NORTH);
        add(getPanel(Color.RED),     BorderLayout.CENTER);
        add(getPanel(Color.MAGENTA), BorderLayout.SOUTH);
        add(getPanel(Color.ORANGE),  BorderLayout.WEST);
        add(getPanel(Color.BLUE),    BorderLayout.EAST);
        pack();
        setLocationRelativeTo(null);
        setVisible(true);
    }
    JPanel getPanel(Color c) {
        JPanel p = new JPanel();
        p.setBackground(c);
        p.setPreferredSize(new Dimension(70, 70));
        return p;
    }
}
```

The program displays



Note that when resizing the window, heights of northern and southern areas do not change and the same applies to widths of west and east areas — the center area consumes all available space.
Note also that you can send only one component to each of the five regions. This is not a problem, however, as you can, as we did in the example above, collect several components in one (very often a JPanel) and send it as single component.

9.5.4 Box layout

The box layout arranges components in one row or one column. Unlike GridLayout, it takes into account the preferred, minimum and maximum sizes of the components, as well as its X- or Y- alignments. In order to set the box layout for a component comp, we call

```
comp.setLayout(new BoxLayout(comp, BoxLayout.X_AXIS); // horizontal
comp.setLayout(new BoxLayout(comp, BoxLayout.Y_AXIS); // vertical
```

and to set its alignment

```
comp.setAlignmentX(align);
comp.setAlignmentY(align);
```

where align is of type **float** and may assume, for X-alignment, three values predefined as constants in class **Component**: LEFT_ALIGNMENT (0.0), CENTER_ALIGNMENT (0.5) and RIGHT_ALIGNMENT (1.0); for Y-alignment these are TOP_ALIGNMENT (0.0), CENTER_ALIGNMENT (0.5) and BOTTOM_ALIGNMENT (1.0). Components will be arranged in the order they have been added to a container (left to right or top to bottom). One can also add special 'filling' components between, below or above them (leftmost and rightmost area for for horizontal boxes). One of these 'fillers' is a rigid area which can be added by invoking

```
comp.add(Box.createRigidArea(new Dimension(x,y));
```

It represents fixed-sized gap between components: it will not change its size when the window is resized. On the other hand, one can add a 'glue' components which will all shrink or expand in the same way when resizing the window.

```
comp.add(Box.createGlue());
```

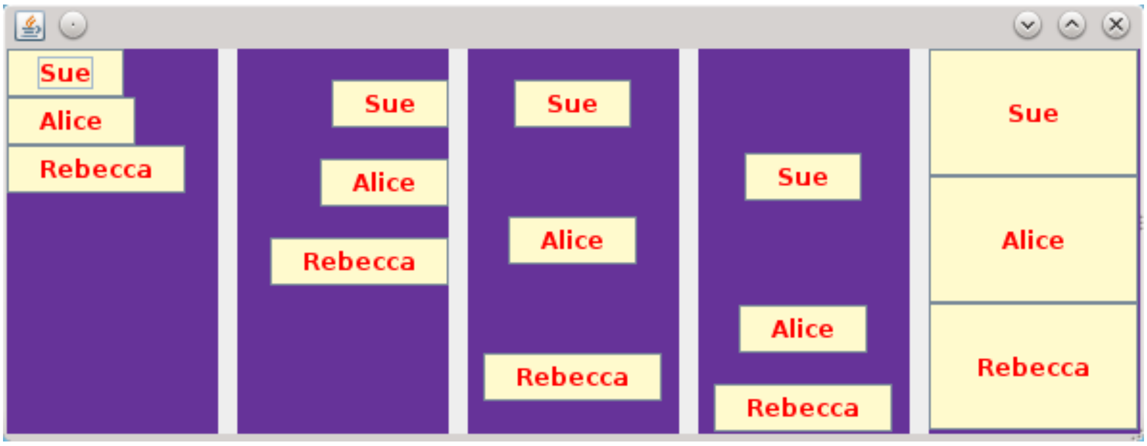
Let us see an example:

Listing 83 MCQ-BoxLay/Boxes.java

```java
import java.awt.Color;
import java.awt.Component;
import java.awt.Dimension;
import java.awt.GridLayout;
import javax.swing.Box;
import javax.swing.BoxLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import static java.awt.Component.LEFT_ALIGNMENT;
import static java.awt.Component.CENTER_ALIGNMENT;

public class Boxes extends JFrame {
    public static void main (String[] args) {
        new Boxes();
    }
    private Boxes() {
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setLayout(new GridLayout(1,0,10,5));
        add(new MyBox(0));
        add(new MyBox(1));
        add(new MyBox(2));
        add(new MyBox(3));
        add(new MyBox(4));
        pack();
        setLocationRelativeTo(null);
        setVisible(true);
    }
}

```

```
class MyBox extends JPanel {
    private static final Color rebecca =
                            new Color(0x66,0x33,0x99);
    private static final Color lfore = Color.RED;
    private static final Color lback =
                            new Color(0xFF,0xFA,0xCD);
    private static final int xsize = 110, ysize = 200;
    private static final float[] align =
        {LEFT_ALIGNMENT, RIGHT_ALIGNMENT, CENTER_ALIGNMENT,
         CENTER_ALIGNMENT, CENTER_ALIGNMENT};
    private static final Dimension rig =
                                new Dimension(0,ysize/12);
    String small = "Sue", medium = "Alice", big = "Rebecca";

    MyBox(int num) {
        setLayout(new BoxLayout(this, BoxLayout.Y_AXIS));
        setPreferredSize(new Dimension(xsize,ysize));
        setBackground(rebecca);

          // before the small button
        switch (num) {
            case 0:                                break;
            case 1:
            case 2: add(Box.createRigidArea(rig)); break;
            case 3: add(Box.createGlue());         break;
            case 4:                                break;
        }

        add(addButton(small, num));

          // between the small and medium buttons
        switch (num) {
            case 0:                                break;
            case 1: add(Box.createRigidArea(rig)); break;
            case 2:
            case 3: add(Box.createGlue());         break;
            case 4:                                break;
        }

        add(addButton(medium, num));

          // between the medium and big buttons
        switch (num) {
            case 0:                                break;
            case 1: add(Box.createRigidArea(rig)); break;
            case 2: add(Box.createGlue());         break;
            case 3: add(Box.createRigidArea(rig)); break;
            case 4:                                break;
        }

        add(addButton(big, num));

          // after the big button
        switch (num) {
            case 0:
            case 1:                                break;
            case 2: add(Box.createRigidArea(rig)); break;
            case 3:
            case 4:                                break;
        }
    }

    private JButton addButton(String txt, int num) {
        JButton but = new JButton(txt);
        but.setForeground(lfore);
        but.setBackground(lback);
        but.setOpaque(true);
        but.setAlignmentX(align[num]);
        if (num == 4)
            but.setMaximumSize(new Dimension(xsize,ysize));
        return but;
    }
}
```

The program displays



Notice, that if the maximum size is not defined or is sufficiently big, the component will occupy the whole available area (see the last column in the figure above).

9.5.5 GridBagLayout

Layout of type GridBagLayout, as those of type GridLayout, divide the area of the component with this layout into rectangular grid of cells. However, unlike GridLayout, it allows to put components into subareas consisting of a rectangular set of neighbouring cells.

Suppose comp is a component with GridBagLayout installed:

```
comp.setLayout(new GridBagLayout());
```

To add a component into comp, we need an object of type GridBagConstraints which carries information where and how this component is to be added. So we create object of type GridBagConstraints, configure it, and then we pass this object when adding components to comp:

```
comp.setLayout(new GridBagLayout);
GridBagConstraints  cnstr = new GridBagConstraints();
  // configuring cnstr
comp.add(anotherComponent, cnstr);
```

Objects of type GridBagConstraints expose several public, modifiable fields, so configuring them consists of a series of assignments. For example

```
GridBagConstraints c = new GridBagConstraints();
c.fill = GridBagConstraints.BOTH;
c.gridx = 0;
c.gridy = 1;
c.weightx = 0.5; // important when resizing
c.weighty = 0.5;
c.gridheight = 2;
// ...
```

Some of the most important fields that we can set are:

- gridx, gridy — specify the x- and y-coordinates of the upper-left cell of the component's display area; coordinates are counted from zero to the right for x-coordinate and downwards for y-coordinate.
- gridwidth, gridheight — specifies the number of columns (gridwidth) and rows (gridheight) in the component's display area. The default values are 1, what corresponds to one cell at position specified by gridx and gridy. One can use GridBagConstraints.REMAINDER to specify that the component's display area will be from gridx to the last cell in the row (for gridwidth) or from gridy to the last cell in the column (for gridheight).
- fill — will be used if the component's display area (a cell or rectangular group of cells) is larger than the component's size. Possible values are defined in GridBagConstraints as constants NONE (leave the size of the components as is, the default), HORIZONTAL (the component is resized to fill its display area horizontally), VERTICAL (the component fills its display area vertically) and BOTH (the component fills its display area in both directions).
- ipadx, ipady — specify the padding around the component (in pixels). By setting non-zero values, we can make some cells larger; remember, however, that all cells of one row have always the same height and all cells in one column have the same width — setting non-zero padding for one cell, therefore, affects widths and heights of other cells.
- weightx, weighty — determine how to distribute space occupied by rows and columns when the whole grid is resized. If the values are 0 (which is the default), the grid will not be rescaled, but will be displayed in the center of the surrounding component. The values of this fields are of type double and lie in the range $[0, 1]$. Their rations determine how rows and columns are scaled. Larger values indicate that the component's row (or column) should get more space when resizing. For each column, its (horizontal) weight corresponds to the highest weightx of its cells; for each row its (vertical) weight is determined by the highest weighty of

its cells. Therefore, to make all cells scale uniformly, one can assign the same non-zero value (0.5, say) to weighty of all cells in one column, and to weightx of all cells in one row.

For the example of using the **GridBagLayout**, see the program in Listing 96 on page 160.

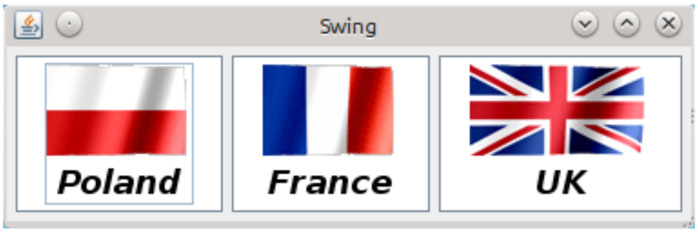
9.6 Using icons

In the next example, we show how to create icons from an existing graphic files:

Listing 84 MBA-IntroSwing/IntroSwing.java

```java
package intro;

import java.awt.Color;
import java.awt.FlowLayout;
import java.awt.Font;
import javax.swing.Icon;
import javax.swing.ImageIcon;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.SwingConstants;
import javax.swing.SwingUtilities;

class IntroSwing  {

    public static void main(String[] args)  {
        SwingUtilities.invokeLater(() -> createGUI());
    }

    private static void createGUI()  {
        Class<IntroSwing> clz = IntroSwing.class;

          // Icons from directory img of the application
        Icon[] icon = {
              // root '/' is dir containing 'intro' package
            new ImageIcon(clz.getResource("/img/pl.gif")),
              // or relative to .class file
            new ImageIcon(clz.getResource("../img/fr.gif")),
            new ImageIcon(clz.getResource("/img/uk.gif")),
        };
          // text on buttons
        String[] descr = {"Poland", "France", "UK" };

        JFrame frame = new JFrame("Swing"); // main window
        frame.setLayout(new FlowLayout());  // layout of its
                                            // contentPane
        for (int i=0; i < icon.length; ++i) {
            JButton b = new JButton(descr[i], icon[i]);
            b.setFont(new Font("Dialog",
                         Font.BOLD | Font.ITALIC, 18));
            b.setForeground(Color.BLACK);
            b.setBackground(Color.WHITE);
              //  position of text relative to icon
            b.setVerticalTextPosition(
                    SwingConstants.BOTTOM);
            b.setHorizontalTextPosition(
                    SwingConstants.CENTER);
            frame.add(b);
        }
        frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        frame.pack();
        frame.setLocationRelativeTo(null);
        frame.setVisible(true);
    }
}
```

The program displays



9.7 *Drawing*

All swing components inherit the method **paintComponent** from **JComponent**. We never call it directly — it is invoked automatically when a component must be repainted (e.g., after resizing, when it is exposed after being hidden behind other windows etc.). There are also two other functions which will be invoked automatically when a component is repainted: **paintChildren** and **paintBorder**, but we seldom have to override them. All these methods take as the argument a so called **graphic context** — object of class **Graphics** from ***java.awt*** (in fact **Graphics2D** extending **Graphics**). It represents, in a sense, an output device on which we can draw geometrical figures or strings — normally, whatever we paint on it, will appear on the component (but can be redirected to memory or a file). When specifying positions of graphic elements, we have to remember that the point with coordinates $(0, 0)$ is in the upper-left corner, and y-coordinate increases *downwards!*

Object of type **Graphics** allow us to:

- set properties of the graphic context;
- draw lines and simple geometrical figures;
- draw strings;
- insert pictures and images (objects of type **Image**).

Geometrical figures can be drawn by invoking

- `void drawLine(int x1, int y1, int x2, int y2)` — draws a line between (x_1, y_1) and (x_2, y_2);

- `void drawOval(int x, int y, int width, int height)` — draws an ellipse (circle) inscribed in a rectangle with upper-left vertex at (x, y) and given width and height;
- `void drawRect(int x, int y, int width, int height)` — draws a rectangle with upper-left vertex at (x, y) and given width and height.

Analogously, we can fill ovals and rectangles by

- `void fillOval(int x, int y, int width, int height)` — fills an ellipse (in parcular a circle) inscribed in a rectangle with upper-left vertex at (x, y) and given width and height;
- `void fillRect(int x, int y, int width, int height)` — fills a rectangle with upper-left vertex at (x, y) and given width and height.

Before each drawing or filling, the color can be changed by **setColor**; if not set, the current foreground color will be used.

The coordinates of pixels should be understood as coordinates of points *between* pixels; when we paint a pixel with coordinates (x, y), the pixel to the right and below this point is painted. For example, to draw a diagonal, we should invoke

```
drawLine(0, 0, getWidth()-1, getHeight()-1);
```

because the pixel referred to by coordinates (width, height) would be outside of the picture! On the other hand, when we fill an oval or a rectangle, pixels *inside* a region specified by coordinates is painted, so to fill the whole area of a component, we would invoke

```
fillRect(0, 0, getWidth(), getHeight());
```

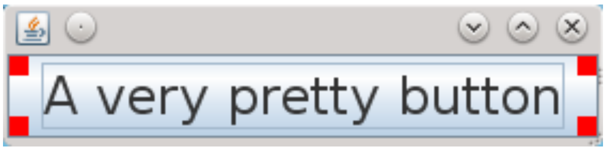
Let us see an example: we draw small squares at the corners of a button. Notice that

> when overriding the **paintComponent** function, it is necessary to invoke, in the first line, the same method from the superclass.

Listing 85 MBB-PrettyButton/PrettyButton.java

```java
import java.awt.Color;
import java.awt.Font;
import java.awt.Graphics;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.SwingUtilities;

class MyButton extends JButton {
    public MyButton(String txt) {
        super(txt);
        setFont(new Font("Dialog", Font.PLAIN, 24));
    }
    @Override
    public void paintComponent(Graphics g) {
        super.paintComponent(g); // IMPORTANT!!!
        int w = getWidth();
        int h = getHeight();
        g.setColor(Color.red);
          // drawing the squares
        g.fillRect(0, 0, 10, 10);
        g.fillRect(w-10, 0, 10, 10);
        g.fillRect(0, h-10, 10, 10);
        g.fillRect(w-10, h-10, 10, 10);
    }
}

public class PrettyButton extends JFrame {
    public PrettyButton() {
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        add(new MyButton("A very pretty button"));
        pack();
        setLocationRelativeTo(null);
        setVisible(true);
    }

    public static void main(String args[]) {
        SwingUtilities.invokeLater(() -> new PrettyButton());
    }
}
```

which produces

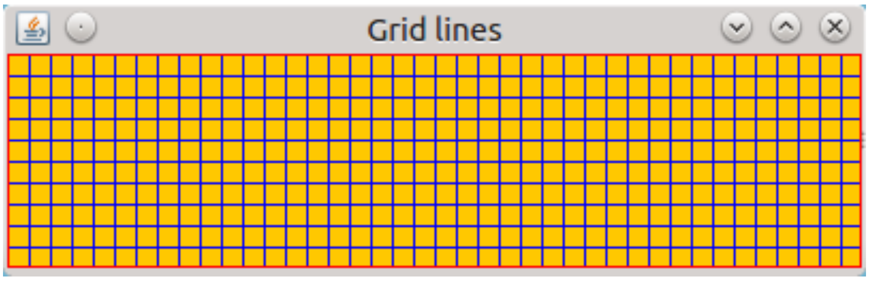


Another example demonstrates how to draw lines and rectangles:

Listing 86 MBE-GridLines/GridLines.java

```
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
import javax.swing.JComponent;
import javax.swing.JFrame;
import javax.swing.SwingUtilities;

public class GridLines extends JFrame {
    public GridLines() {
        super("Grid lines");
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        add(new MyComponent(400, 100));
        pack();
        setLocationRelativeTo(null);
        setVisible(true);
    }

    public static void main(String[] args) {
        SwingUtilities.invokeLater(() -> new GridLines());
    }
}

class MyComponent extends JComponent {
    public MyComponent(int w, int h) {
        Dimension d = new Dimension(w, h);
        setMinimumSize(d);
        setPreferredSize(d);
        setMaximumSize(d);
    }
    @Override
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        int w = getWidth();
        int h = getHeight();
        g.setColor(Color.ORANGE);
        g.fillRect(0, 0, w, h);
        g.setColor(Color.RED);
        g.drawRect(0, 0, w-1, h-1);
        g.setColor(Color.BLUE);
        for (int y = 10; y < h-1; y += 10)
            g.drawLine(1, y, w-2, y);
        for (int x = 10; x < w-1; x += 10)
            g.drawLine(x, 1 , x, h-2);
    }
}
```

The program displays



and the window may be resized.

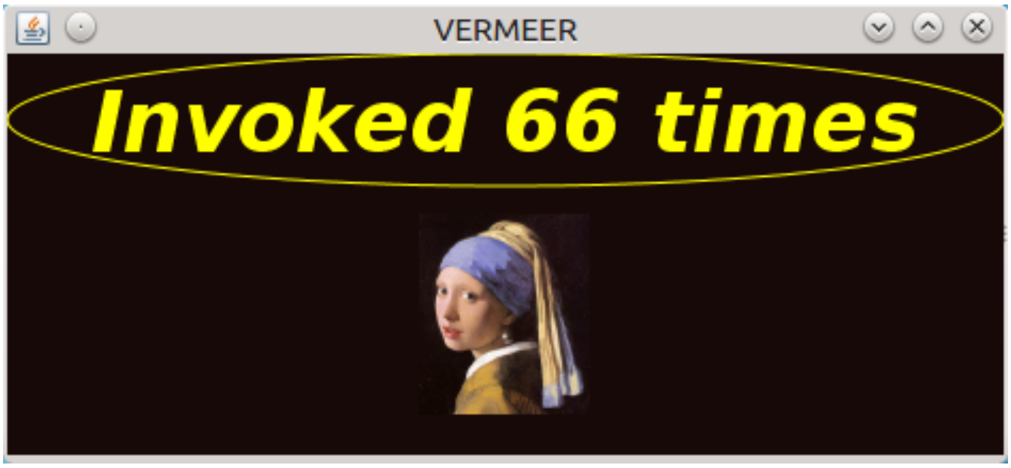
In the next example, we demonstrate how an image can be loaded, scaled and displayed (in fact, it can be done in several ways). The example also shows how to draw a string:

Listing 87 MCM-Drawing/Drawing.java

```java
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.FontMetrics;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Image;
import java.awt.RenderingHints;
import java.awt.image.BufferedImage;
import java.awt.geom.Ellipse2D;
import java.awt.geom.Rectangle2D;
import java.io.File;
import java.io.IOException;
import javax.imageio.ImageIO;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.SwingUtilities;

public class Drawing {
    public static void main (String[] args) {
        SwingUtilities.invokeLater(() -> new Drawing());
    }

    private Drawing() {
        JFrame fr = new JFrame("VERMEER");
        fr.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JPanel panel = new MyPanel();
        //fr.setContentPane(panel);
        fr.add(panel);
        fr.pack();
        fr.setLocationRelativeTo(null);
        fr.setVisible(true);
    }
}

class MyPanel extends JPanel {
    int counter = 0;
    BufferedImage img = null;

    MyPanel() {
        setBackground(new Color(23,9,8));
        setForeground(Color.YELLOW);
        setOpaque(true);
        setFont(new Font("Sans Serif",
                    Font.BOLD | Font.ITALIC,40));
        setPreferredSize(new Dimension(600,350));
        try {
              // loading the image
```

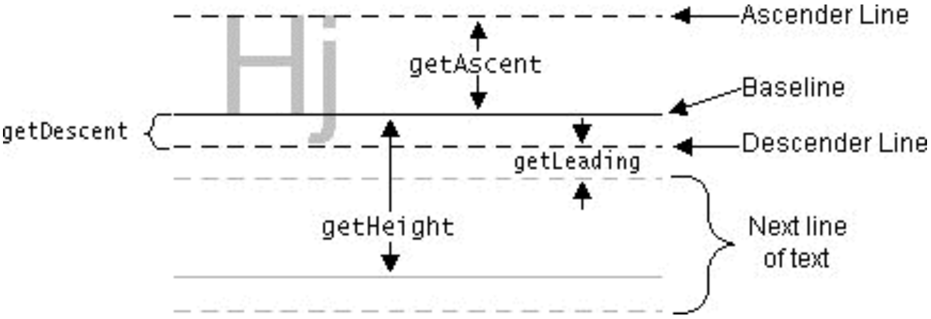
```java
            img = ImageIO.read(new File("vermeer.png"));
        } catch (IOException e) {
            System.out.println("Image file not found");
            System.exit(1);
        }
    }
    @Override
    protected void paintComponent(Graphics g) {
          // not necessary, just to make things prettier
        Graphics2D g2 = (Graphics2D)g;
        g2.setRenderingHint(
                RenderingHints.KEY_STROKE_CONTROL,
                RenderingHints.VALUE_STROKE_PURE);
        g2.setRenderingHint(
                RenderingHints.KEY_ANTIALIASING,
                RenderingHints.VALUE_ANTIALIAS_ON);
        g2.setRenderingHint(
                RenderingHints.KEY_TEXT_ANTIALIASING,
                RenderingHints.VALUE_TEXT_ANTIALIAS_ON);
          // could have been just
          // super.paintComponent(g);
        super.paintComponent(g2);

        String str = "Invoked " + ++counter + " times";
        FontMetrics fm = g2.getFontMetrics();
        Rectangle2D r = fm.getStringBounds(str,g2);

        int w = getWidth();
        int h = getHeight();
        int x = (w-(int)r.getWidth())/2;
        int y = h/6-(int)r.getHeight()/2+fm.getAscent();
        g2.drawString(str, x, y);
        g2.drawOval(0, 0, w, h/3);
        Image im = img.getScaledInstance( -1, h/2,
                               Image.SCALE_SMOOTH);
        g2.drawImage(im,(w-im.getWidth(null))/2,2*h/5,null);
    }
}
```

The program produces



The picture will be rescaled when the window is resized. Note that the counter shows how many times the **paintComponent** function has been invoked.

In the above program we used class **FontMetrics** to get information on properties of the string treated as a graphics; for example we can get sizes of the string rendered in a given font (as an object of type **Rectangle2D**). Strings rendered as a graphics have also other characteristics, shown in the picture



and available by invoking several methods of **FontMetrics**, like **getHeight**, **getAscent**, **getDescent**, **getLeading** and others. Note, that we cast **Graphics** object into **Graphics2D** — this is always safe, because the object passed to **paintComponent** *is* of this type. We did it in order to use methods of **Graphics2D** that are *not* inherited from **Graphics**: those methods yield a better quality of the displayed graphics. The program belows demonstrates the difference:

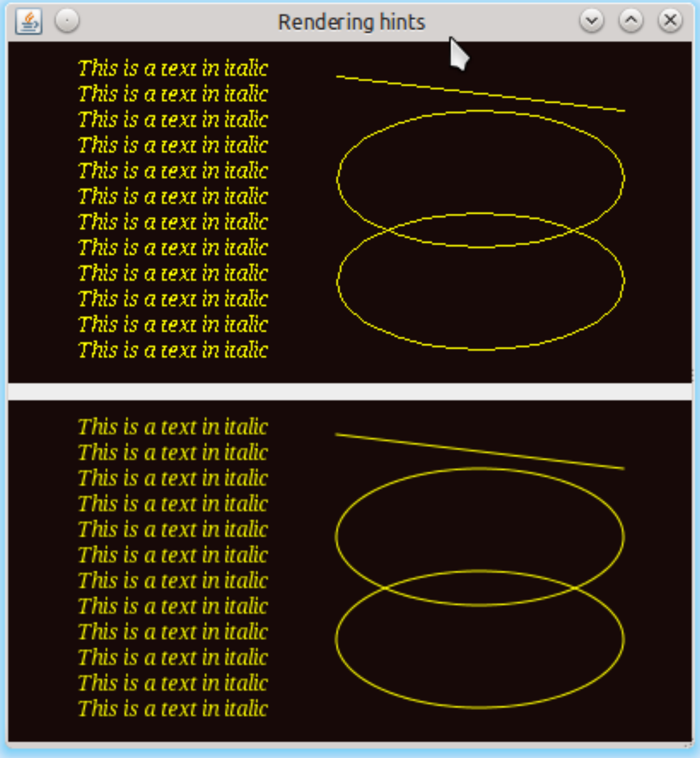
Listing 88 MCY-RenderHints/RenderHints.java

```java
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.FontMetrics;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.GridLayout;
import java.awt.RenderingHints;
import java.awt.geom.Rectangle2D;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.SwingUtilities;

```

```
public class RenderHints {
    public static void main (String[] args) {
        SwingUtilities.invokeLater(() -> new RenderHints());
    }
    private RenderHints() {
        JFrame fr = new JFrame("Rendering hints");
        fr.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        fr.setLayout(new GridLayout(2,1,10,10));
        fr.add(new MyPanel(false));
        fr.add(new MyPanel(true));
        fr.pack();
        fr.setLocationRelativeTo(null);
        fr.setVisible(true);
    }
}

class MyPanel extends JPanel {
    boolean hintsOn;

    MyPanel(boolean hints) {
        hintsOn = hints;
        setBackground(new Color(23,9,8));
        setForeground(Color.YELLOW);
        setOpaque(true);
        setPreferredSize(new Dimension(400,200));
    }
    @Override
    //protected void paintComponent(Graphics g) {
    public void paintComponent(Graphics g) {
        Graphics2D g2 = (Graphics2D)g;
        if (hintsOn) {
            g2.setRenderingHint(
                    RenderingHints.KEY_STROKE_CONTROL,
                    RenderingHints.VALUE_STROKE_PURE);
            g2.setRenderingHint(
                    RenderingHints.KEY_ANTIALIASING,
                    RenderingHints.VALUE_ANTIALIAS_ON);
            g2.setRenderingHint(
                    RenderingHints.KEY_TEXT_ANTIALIASING,
                    RenderingHints.VALUE_TEXT_ANTIALIAS_ON);
        }
        super.paintComponent(g2);

        String str = "This is a text in italic";
        g2.setFont(new Font("Serif",Font.ITALIC,12));
        FontMetrics fm = g2.getFontMetrics();
        Rectangle2D r = fm.getStringBounds(str,g2);

        int w = getWidth();
        int h = getHeight();
```

```
            int x = w/10;
            int y = h/10;
            while (y < 0.95*h) {
                g2.drawString(str, x, y);
                y += (int)(r.getHeight()*1.1);
            }
            x = (int)(2*w/10+r.getWidth());
            y = h/10;
            g2.drawLine(x, y, 9*w/10, 2*h/10);
            g2.drawOval(x, 2*h/10, 9*w/10-x, 4*h/10);
            g2.drawOval(x, 5*h/10, 9*w/10-x, 4*h/10);
        }
    }
```

and the difference becomes apparent



9.8 Windows

Windows are containers of the highest level in hierarchy of containers/components — they are heavyweight and contain all other components (usually lightweight) of the whole GUI.
Some windows may have an owner (they are then called *secondary window*), some — *primary windows* — do not: they are always 'parents' for other containers/components.

- Closing a primary window closes all its children (secondary windows);
- Minimizing a primary window minimizes all its contents;

- Moving a primary window moves it with all its contents.

Only secondary windows can (but need not to) be modal; when a window is modal, interaction with the parent window is blocked until this modal window has been closed.

Let us mention some of the most important methods that we can call on windows:

- `void pack()` — 'packs' the window, i.e., calculates all sizes and locations of the child components taking into account their preferred sizes;
- `void setLocationRelativeTo(Component c)` — set location of this window on a specified component; center of the screen if c is null;
- `setDefaultCloseOperation(int)` specifies what will happen when the window is closed. This is determined by an integer defined in class JFrame: EXIT_ON_CLOSE, DISPOSE_ON_CLOSE, DO_NOTHING_ON_CLOSE or HIDE_ON_CLOSE;
- `Toolkit getToolkit()` — returns Toolkit which contains many useful methods describing the graphical environment of the application. The class constitutes a 'glue' between platform independent classes and native operating system;
- `boolean isShowing()` — tests whether the window is displayed on the screen;
- `void setCursor(Cursor)` — set the type of the cursor;
- `dispose()` — releases resources related to the widow;
- `Window getOwner()` — returns the owner of this window (or null);
- `Window[] getOwnedWindows()` — returns an array of owned (child) windows;
- `Component getFocusOwner()` — returns the component inside the widow that has the focus (if the focus is somewhere in this window);
- `Component getMostRecentFocusOwner()` — returns the component of the window that will receive the focus when the whole window will get it;

> The most important kind of window is the frame window (in Swing it is JFrame) which has borders, title bar, control icons, menu bar, tool bar etc.

A frame window has no owner and cannot be modal. It can be created with the default constructor or with a String argument specifying its title (that can be then dynamically modified).

Frames have many properties that can be get/set by appropriate methods; some of them are

- iconImage (setIconImage/getIconImage) of type Image — icon used on the task bar when the window is minimized;
- menuBar of type JMenuBar;
- title of type String — a string appearing on the title bar of the window;
- resizable of type boolean — specifies, if the window can be resized; may be modified dynamically;
- undecorated of type boolean — if true, the window has no 'decorations' (title bar, borders etc.);
- extendedState of type int — specifies the current state of the window as an integer constant from class JFrame: NORMAL, ICONIFIED, MAXIMIZED_HORIZ, MAXIMIZED_VERT, MAXIMIZED_BOTH. We can check whether a particular state is available on our platform by invoking Toolkit.isFrameStateSupported(int).

A very close cousin of JFrame is JDialog. It is intended to display some sort of a dialog allowing the user, for example, to enter some data. It alway has a parent, can (and usually should) be modal. As it represents some sort of a 'helper' widget, you cannot set its default close operations to EXIT_ON_CLOSE. Such dialogs are often created by invocation of static methods from class JOptionPane (many overloaded versions of showInputDialog or showMessageDialog). Modality can be set by invoking setModalityType — the corresponding property has enum type Dialog.ModalityType; use MODELESS to make the dialog non-modal, DOCUMENT_MODAL to make it modal (its parents will be blocked) or APPLICATION_MODAL (all root windows of an application will be blocked).

There are also windows of type JInternalFrame that are *lightweight*: they always have a parent and are contained inside another container. As they are lightweight, they can be completely platform independent and have additional properties lacking in heavyweight windows.

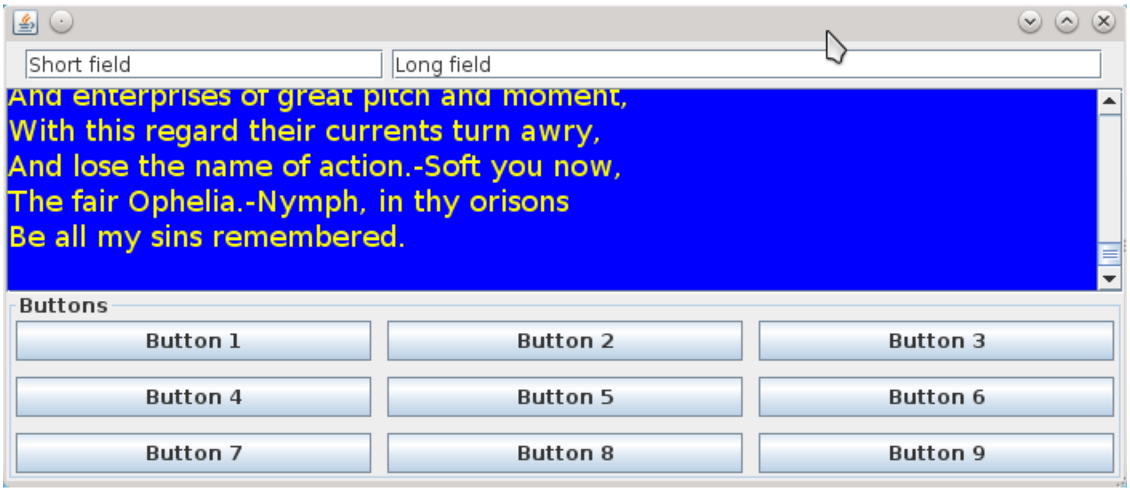
9.9 More examples

The example below illustrates various components, in particular a text area with scroll bars, and also invokeLater which is called from the main thread to modify the displayed GUI (append a line to the text area and possibly add a vertical scroll bar):

Listing 89 MCJ-Layouts1/Layouts.java

```java
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.FlowLayout;
import java.awt.Font;
import java.awt.GridLayout;
import java.io.BufferedReader;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
import javax.swing.BorderFactory;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;
import javax.swing.SwingUtilities;

public class Layouts extends JFrame {
    static JTextArea area = null;

    public static void main (String[] args) {
        new Layouts();

        try (BufferedReader br =
                Files.newBufferedReader(
                    Paths.get("hamlet.txt"))) {
            String line = null;
            while ( (line = br.readLine()) != null ) {
                String s = line;
                SwingUtilities.invokeLater(
                        () -> area.append(s+"\n"));
                Thread.sleep(500);
            }
            SwingUtilities.invokeLater( () -> {
                area.setBackground(Color.BLUE);
                area.setForeground(Color.YELLOW);
            });
        } catch(IOException | InterruptedException e) {
            e.printStackTrace();
            return;
        }
    }

    private Layouts() {
        setDefaultCloseOperation(EXIT_ON_CLOSE);
          // setting layout of the whole contentPane
          // of the frame window (it is border layout
          // by default anyway...)
        setLayout(new BorderLayout());

        JPanel southPanel = new JPanel();
        southPanel.setBorder(
            BorderFactory.createTitledBorder("Buttons"));
        southPanel.setLayout(new GridLayout(3,3,10,10));
        for (int i = 1; i < 10; ++i)
            southPanel.add(new JButton("Button " + i));
        add(southPanel,BorderLayout.SOUTH);

        JPanel northPanel = new JPanel();
        northPanel.setLayout(
                        new FlowLayout(FlowLayout.CENTER));
        northPanel.add(new JTextField("Short field",20));
        northPanel.add(new JTextField("Long field",40));
        add(northPanel,BorderLayout.NORTH);

        area = new JTextArea(15,40);
        area.setFont(
            new Font("Sans_Serif",Font.PLAIN,18));
        area.setBackground(Color.WHITE);
        area.setForeground(Color.BLACK);
        JScrollPane scroll = new JScrollPane(area,
                JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
                JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED);
        add(scroll,BorderLayout.CENTER);

        pack();
        setLocationRelativeTo(null);
        setVisible(true);
    }
}
```

The program displays



Short field

Long field

```
And enterprises of great pitch and moment,
With this regard their currents turn awry,
And lose the name of action.-Soft you now,
The fair Ophelia.-Nymph, in thy orisons
Be all my sins remembered.
```

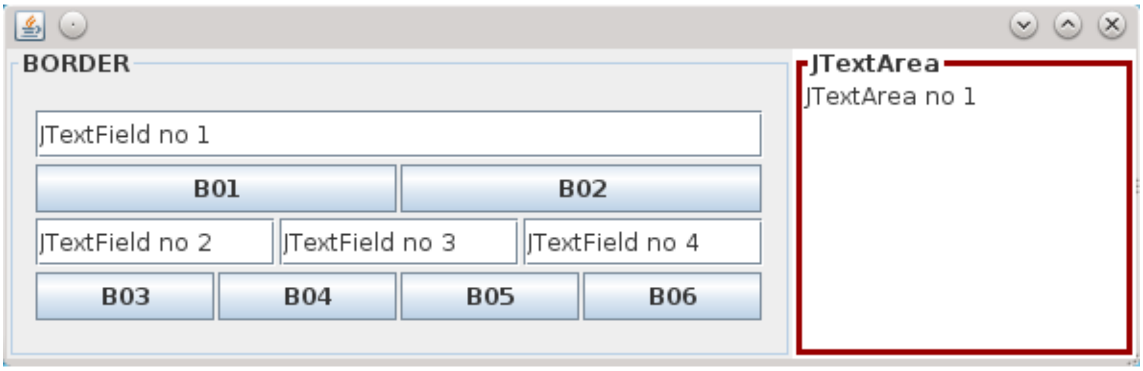
Another example is similar: here we create more complex borders, in particular compound borders:

Listing 90 MCK-Layouts/Layouts.java

```
import java.awt.Color;
import java.awt.BorderLayout;
import java.awt.GridLayout;
import javax.swing.BorderFactory;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JTextArea;
import javax.swing.JTextField;

public class Layouts extends JFrame {
    public static void main(String[] args) {
        new Layouts();
    }

    Layouts() {
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setContentPane(new MainPanel());
        pack();
        setLocationRelativeTo(null);
        setVisible(true);
    }
}
```

```java

class MainPanel extends JPanel {
    MainPanel() {
        JTextField[] tft = new JTextField[4];
        for (int i = 0; i < tft.length; i++) {
            tft[i]=new JTextField(11);
            tft[i].setText("JTextField no "+(i+1));
        }

        JButton[] bt = new JButton[6];
        for (int i = 0; i < bt.length; i++)
            bt[i]=new JButton(String.format("B%02d",i+1));

        JPanel[] panels = new JPanel[6];
        for (int i = 0; i < panels.length; i++)
            panels[i] = new JPanel();

        JTextArea tat = new JTextArea(3,15);
        tat.setText("JTextArea no 1");
        tat.setBorder(
            BorderFactory.createTitledBorder(
                BorderFactory.createLineBorder(
                    new Color(0x99,0,0),3
                ),
                "JTextArea"
            )
        );

        // two buttons in a row
        panels[0].setLayout(new GridLayout(1,2,2,2));
        panels[0].add(bt[0]);
        panels[0].add(bt[1]);

        // four buttons in a row
        panels[1].setLayout(new GridLayout(1,4,2,2));
        for (int i = 2; i < 6; i++)
            panels[1].add(bt[i]);

        // one text field in a row
        panels[2].setLayout(new BorderLayout());
        panels[2].add(tft[0],BorderLayout.CENTER);

        // three text fields in a row
        panels[3].setLayout(new GridLayout(1,3,2,2));
        for (int i = 1; i < 4; i++)
            panels[3].add(tft[i]);

        // text area
        panels[4].setLayout(new BorderLayout());
        panels[4].add(tat,BorderLayout.CENTER);

        // buttons and text fields
        panels[5].setLayout(new GridLayout(4,1,5,3));
        panels[5].add(panels[2]);
        panels[5].add(panels[0]);
        panels[5].add(panels[3]);
        panels[5].add(panels[1]);
        panels[5].setBorder(
            BorderFactory.createCompoundBorder(
                  // outer
                BorderFactory.createTitledBorder("BORDER"),
                  // inner
                BorderFactory.createEmptyBorder(15,10,15,10)
            )
        );

        setLayout(new BorderLayout());
        add(panels[5], BorderLayout.CENTER);
        add(panels[4], BorderLayout.EAST);
    }
}
```

The program displays



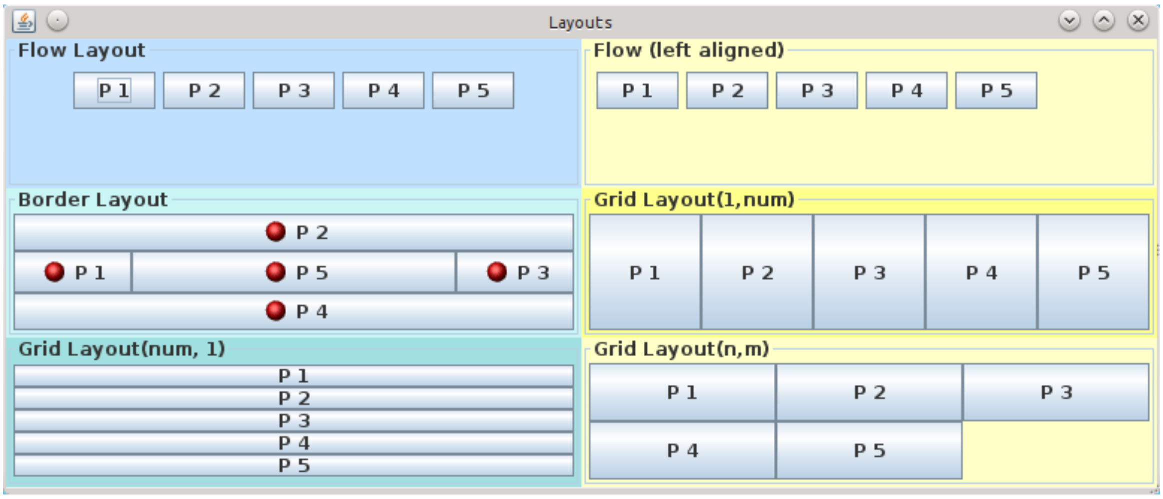
In the example below, we build even more complex GUI

Listing 91 MCL-Layouts2/MiscLayouts.java

```
import java.awt.BorderLayout;
import java.awt.FlowLayout;
import java.awt.GridLayout;
import java.awt.LayoutManager;
import java.awt.Color;
import javax.swing.BorderFactory;
import javax.swing.Icon;
import javax.swing.ImageIcon;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
```

```java

public class MiscLayouts {
    public static void main(String[] args) {
          // number of comonents in panels
        final int CNUM = 5;
          // descriptions
        String lmNames[] = {
            "Flow Layout", "Flow (left aligned)",
            "Border Layout", "Grid Layout(1,num)",
            "Grid Layout(num, 1)", "Grid Layout(n,m)"
        };
          // layouts
        LayoutManager lm[] = {
            new FlowLayout(),
            new FlowLayout(FlowLayout.LEFT),
            new BorderLayout(),
            new GridLayout(1, 0),
            new GridLayout(0, 1),
            new GridLayout(2, 0)
        };

          // for BorderLayout
        String gborders[] = {
                BorderLayout.WEST,
                BorderLayout.NORTH,
                BorderLayout.EAST,
                BorderLayout.SOUTH,
                BorderLayout.CENTER
        };
          // panel colors
        Color colors[] = {
            new Color(191, 225, 255),
            new Color(255, 255, 200),
            new Color(201, 245, 245),
            new Color(255, 255, 140),
            new Color(161, 224, 224),
            new Color(255, 255, 200)
        };

          // icon on a button
        Icon redDot = new ImageIcon("red.gif");

        JFrame frame = new JFrame("Layouts");
        frame.setLayout(new GridLayout(0, 2));

        for (int i = 0; i < lmNames.length; i++) {
            JPanel p = new JPanel();
            p.setBackground(colors[i]);
            p.setBorder(BorderFactory
                       .createTitledBorder(lmNames[i]));
            p.setLayout(lm[i]);
            Icon icon = null;

            if (lm[i] instanceof BorderLayout) icon = redDot;
            for (int j = 0; j < CNUM; j++) {
                JButton b = new JButton("P " + (j+1), icon);
                p.add(b, gborders[j]);
            }
            frame.add(p);
        }
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.pack();
        frame.setLocationRelativeTo(null);
        frame.setVisible(true);
    }
}
```

The program displays

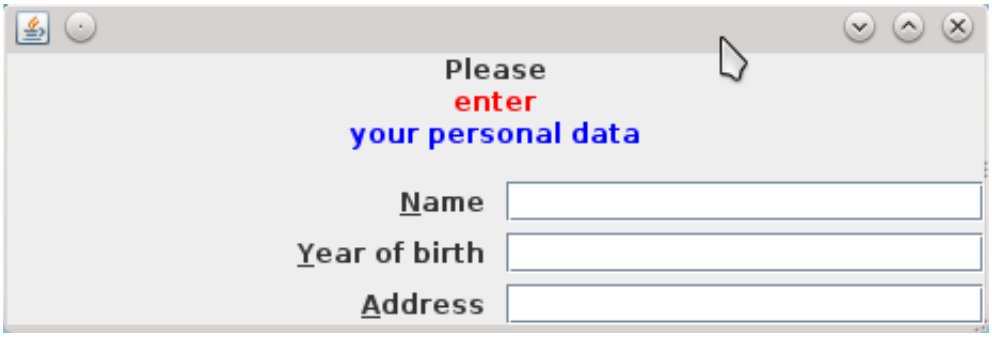


The next example shows how to set mnemonics and how to connect a label with another component:

Listing 92 MCH-LabsFor/LabelsFor.java

```java
import java.awt.BorderLayout;
import java.awt.GridLayout;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;

class LabelsFor extends JFrame {
    JPanel panel = new JPanel(new GridLayout(0, 2, 10, 5));

    public static void main(String args[]) {
        new LabelsFor();
    }

    LabelsFor() {
        setLayout(new BorderLayout()); // not needed here
        String html = "<html><center>Please<br>"
            + "<b><font color=red>enter</font></b><br>"
            + "<font color=blue>your personal data"
            + "</font></center><br></html>";
        JLabel head = new JLabel(html, JLabel.CENTER);
        add(head, BorderLayout.NORTH);
        addLabAndTxtFld("Name", 'n', "Enter your name");
        addLabAndTxtFld("Year of birth", 'y', "YYYY/MM/DD");
        addLabAndTxtFld("Address", 'a', "Your address");
        add(panel, BorderLayout.CENTER);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        pack();
        setLocationRelativeTo(null);
        setVisible(true);
    }

    void addLabAndTxtFld(String txt, char mnem, String tip){
        JLabel lab = new JLabel(txt, JLabel.RIGHT);
        JTextField tf = new JTextField(20);
        tf.setToolTipText(tip);
        lab.setLabelFor(tf);
        lab.setDisplayedMnemonic(mnem);
        panel.add(lab);
        panel.add(tf);
    }
}
```

The program displays



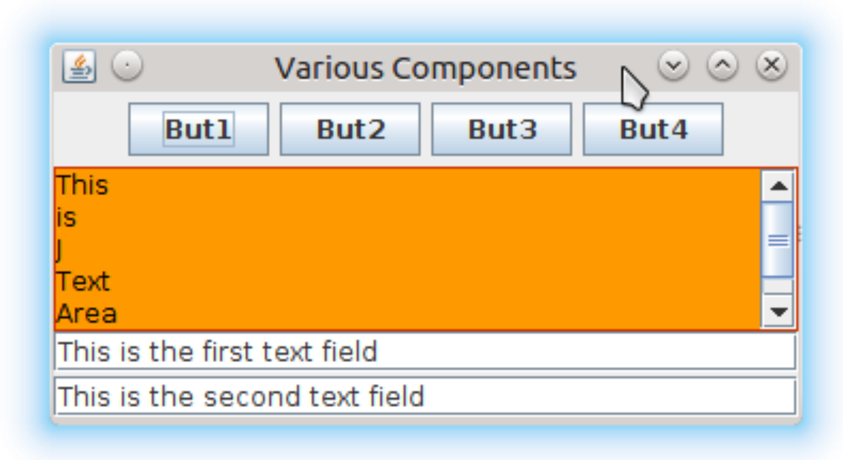
And one more example with JScrollPane:

Listing 93 MCG-Components/VariousComponents.java

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.FlowLayout;
```

```java
import java.awt.GridLayout;
import javax.swing.BorderFactory;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;
import javax.swing.SwingUtilities;

public class VariousComponents extends JFrame {
    public static void main(String[] args) {
        new VariousComponents();
    }

    VariousComponents() {
        super("Various Components");
        setDefaultCloseOperation(EXIT_ON_CLOSE);

          // this is the default anyway
        setLayout(new BorderLayout());

        JPanel lower = new JPanel();
        lower.setLayout(new GridLayout(2,1,5,2));
        JTextField tup = new JTextField(30);
        JTextField tdn = new JTextField(30);
        tup.setText("This is the first text field");
        tdn.setText("This is the second text field");
        lower.add(tup);
        lower.add(tdn);
        add(lower,BorderLayout.SOUTH);

        JPanel upper = new JPanel();
        upper.setLayout(new FlowLayout());
        JButton b1 = new JButton("But1");
        JButton b2 = new JButton("But2");
        JButton b3 = new JButton("But3");
        JButton b4 = new JButton("But4");
        upper.add(b1);
        upper.add(b2);
        upper.add(b3);
        upper.add(b4);
        add(upper,BorderLayout.NORTH);

        JTextArea ja = new JTextArea(5,30);
        ja.setBackground(new Color(255,153,0));
        ja.setForeground(Color.BLACK);
        ja.setText("This\nis\nJ\nText\nArea\n!");
        JScrollPane sc = new JScrollPane(ja,
                JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
                JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED);
        sc.setBorder(
            BorderFactory.createLineBorder(
                                      new Color(204,51,0)));
        add(sc,BorderLayout.CENTER);

        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                pack();
                setLocationRelativeTo(null);
                setVisible(true);
            }
        });
    }
}
```

The program displays



This
is
J
Text
Area

This is the first text field

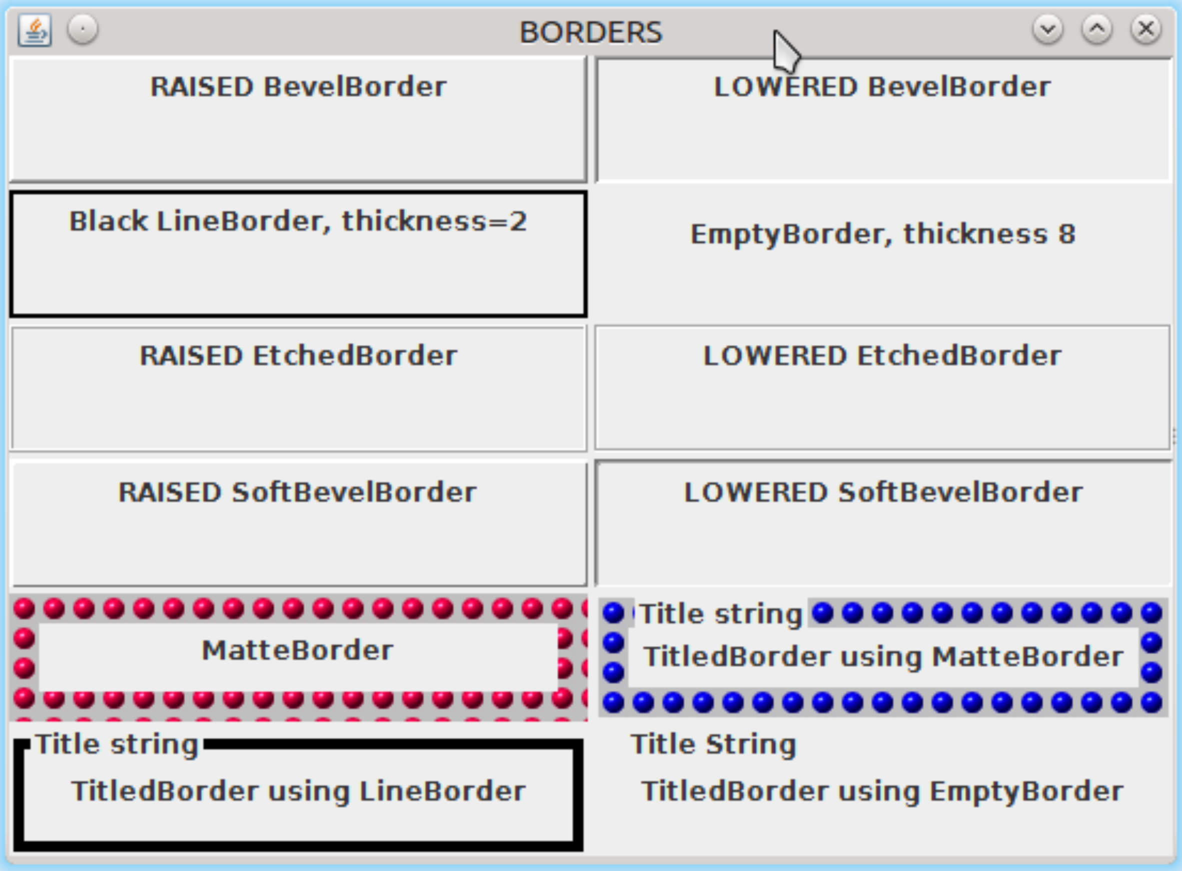
This is the second text field

The next example just shows various borders:

Listing 94 MCP-Borders/Borders.java

```
import java.awt.Color;
import java.awt.GridLayout;
import javax.swing.ImageIcon;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.border.BevelBorder;
import javax.swing.border.EmptyBorder;
import javax.swing.border.EtchedBorder;
import javax.swing.border.LineBorder;
import javax.swing.border.MatteBorder;
import javax.swing.border.SoftBevelBorder;
import javax.swing.border.TitledBorder;

public class Borders extends JFrame {
    public static void main(String args[]) {
        new Borders();
    }

    public Borders() {
        super("BORDERS");
        setDefaultCloseOperation(EXIT_ON_CLOSE);

        JPanel content = new JPanel();
        content.setLayout(new GridLayout(6,2,3,3));

        JPanel p = new JPanel();
        p.setBorder(new BevelBorder(BevelBorder.RAISED));
        p.add(new JLabel("RAISED BevelBorder"));
        content.add(p);

        p = new JPanel();
        p.setBorder(new BevelBorder(BevelBorder.LOWERED));
        p.add(new JLabel("LOWERED BevelBorder"));
        content.add(p);

        p = new JPanel();
        p.setBorder(new LineBorder(Color.black, 2));
        p.add(new JLabel("Black LineBorder, thickness=2"));
        content.add(p);

        p = new JPanel();
        p.setBorder(new EmptyBorder(8, 8, 8, 8));
        p.add(new JLabel("EmptyBorder, thickness 8"));
        content.add(p);

        p = new JPanel();
        p.setBorder(new EtchedBorder(EtchedBorder.RAISED));
        p.add(new JLabel("RAISED EtchedBorder"));
        content.add(p);

        p = new JPanel();
        p.setBorder(new EtchedBorder(EtchedBorder.LOWERED));
        p.add(new JLabel("LOWERED EtchedBorder"));
        content.add(p);

        p = new JPanel();
        p.setBorder(new SoftBevelBorder(
                    SoftBevelBorder.RAISED));
        p.add(new JLabel("RAISED SoftBevelBorder"));
        content.add(p);

        p = new JPanel();
        p.setBorder(new SoftBevelBorder(
                    SoftBevelBorder.LOWERED));
        p.add(new JLabel("LOWERED SoftBevelBorder"));
        content.add(p);

        p = new JPanel();
        p.setBorder(new MatteBorder(
                    new ImageIcon("redball.gif")));
        p.add(new JLabel("MatteBorder"));
        content.add(p);

        p = new JPanel();
        p.setBorder(new TitledBorder(new MatteBorder(
                        new ImageIcon("blueball.gif")),
                          "Title string"));
        p.add(new JLabel("TitledBorder using MatteBorder"));
        content.add(p);

        p = new JPanel();
        p.setBorder(new TitledBorder(
                        new LineBorder(Color.black, 5),
                          "Title string"));
        p.add(new JLabel("TitledBorder using LineBorder"));
        content.add(p);

        p = new JPanel();
        p.setBorder(new TitledBorder(
                        new EmptyBorder(10, 10, 10, 10),
                          "Title String"));
        p.add(new JLabel("TitledBorder using EmptyBorder"));
        content.add(p);

        add(content);
        pack();
        setLocationRelativeTo(null);
        setVisible(true);
    }
}
```

The program displays



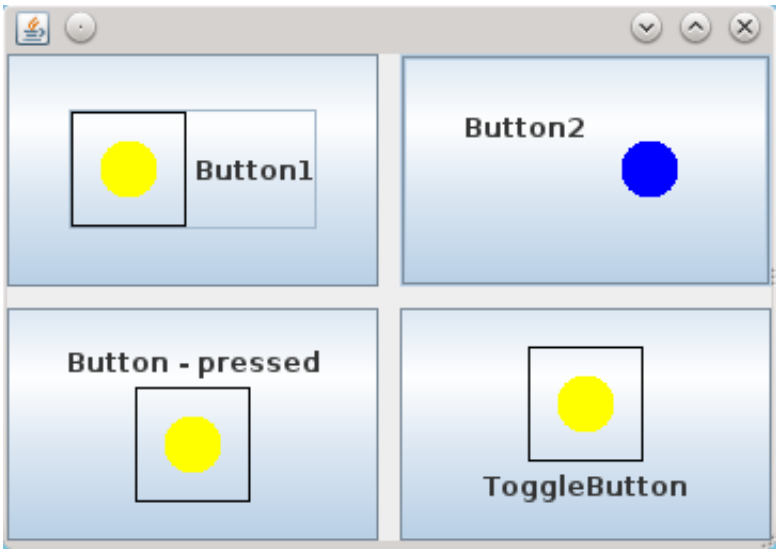
In the following example, we demonstrate, how one can set icons for buttons, different for different states of the button. Moreover, icons are custom made — we draw them manually, so we don't need any additional graphic files. Components JToggleButton are by themselves not very interesting, but the class is the base for much more useful JRadioButton and JCheckBox.

Listing 95 MCU-Buttons/Buttons.java

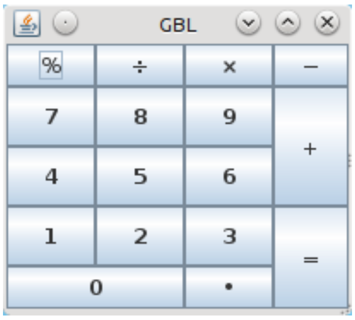
```java
import java.awt.Color;
import java.awt.Component;
import java.awt.Graphics;
import java.awt.GridLayout;
import javax.swing.AbstractButton;
import javax.swing.Icon;
import javax.swing.JButton;
import javax.swing.JComponent;
import javax.swing.JFrame;
import javax.swing.JToggleButton;
import javax.swing.SwingUtilities;
import static javax.swing.SwingConstants.*;


public class Buttons extends JFrame {
    private Icon[] icons = {
        new MyIcon(Color.YELLOW, true),
        new MyIcon(Color.BLUE, false),
        new MyIcon(Color.RED, true),
        new MyIcon(Color.BLACK, false)
    };

    // will be programatically pressed
    private JButton bpre = new JButton("Button - pressed");

    public static void main(String args[]) {
        new Buttons();
    }

    Buttons() {
        setLayout(new GridLayout(2, 2, 10, 10));
        JButton b = new JButton("Button1");
        setButt(b, icons, RIGHT, CENTER);
        JButton bmov = new JButton("Button2");
        setButt(bmov, icons, LEFT, TOP);
        setButt(bpre, icons, CENTER, TOP);

        JToggleButton tb = new JToggleButton(
                "ToggleButton");
        setButt(tb, icons, CENTER, BOTTOM);

        setDefaultCloseOperation(DISPOSE_ON_CLOSE);
        pack();
        setLocationRelativeTo(null);
        setVisible(true);
        try {
            Thread.sleep(2500);
            SwingUtilities.invokeLater(() -> {
                bpre.doClick(500); // pressed for 500 ms
            });
        } catch(Exception ignore) { }
    }

    void setButt(AbstractButton b, Icon[] i,
            int horPos, int vertPos) {
        b.setFocusPainted(true);
        b.setIcon(i[0]);
        b.setRolloverIcon(i[1]);
        b.setPressedIcon(i[2]);
        b.setSelectedIcon(i[3]);
        b.setHorizontalTextPosition(horPos);
        b.setVerticalTextPosition(vertPos);
        add(b);
    }
}
```

```

class MyIcon implements Icon {

    private Color color;
    private int w = 80;
    private boolean frame;

    MyIcon(Color c, boolean frame) {
        color = c;
        this.frame = frame;
    }

    @Override
    public void paintIcon(Component c,
            Graphics g, int x, int y) {
        Color old = g.getColor();
        g.setColor(color);
        w = ((JComponent) c).getHeight()/2;
        int p = w/4, d = w/2;
        g.fillOval(x + p, y + p, d, d);
        if (frame) {
            g.setColor(Color.BLACK);
            g.drawRect(x, y, w-1, w-1);
        }
        g.setColor(old);
    }

    @Override
    public int getIconWidth()  { return w; }
    @Override
    public int getIconHeight() { return w; }
}
```

The program displays



The last example demonstrates **GridBagLayout** applied to a calculator-like application, which displays the following interface



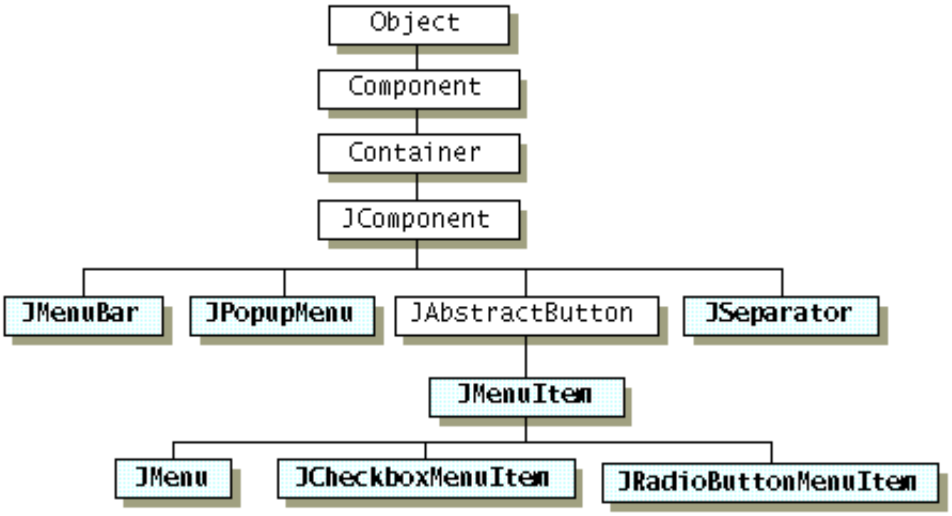
The code is presented below. Note that for each component added to the main **JPanel**, we create a separate object **GridBagConstraints**. This is not needed, we could reuse existing objects for several components. However, such approach can be rather error-prone, as we would have to remember which fields are set and reset them to other values before applying to another components.

Listing 96 MDB-GridBag/GridBag.java

```java
import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class GridBag {
    public static void main (String[] args) {
        JFrame f = new JFrame("GBL");
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f.add(new MyPanel());
        f.pack();
        f.setLocationRelativeTo(null);
        f.setVisible(true);
    }
}

class MyPanel extends JPanel {
    MyPanel() {
        setLayout(new GridBagLayout());
          // upper row
        String[] ur = {"%","\u00f7","\u00d7","\u2212"};
        for (int i = 0; i < ur.length; ++i) {
            GridBagConstraints c = new GridBagConstraints();
            c.fill = GridBagConstraints.BOTH;
            c.gridx = i;
            c.gridy = 0;
            c.weightx = 0.5; // important when resizing
            c.weighty = 0.5;
            add(new JButton(ur[i]),c);
        }
          // numeric pad
        for (int i = 1; i <= 9; ++i) {
            GridBagConstraints c = new GridBagConstraints();
            c.fill = GridBagConstraints.BOTH;
            c.gridx = (i-1)%3;
            c.gridy = 3 - (i-1)/3;
            c.weighty = 0.5;
            c.ipadx = 10; // digit buttons will be larger
            c.ipady = 10;
            add(new JButton(""+i),c);
        }
          // plus and == at rhs
        String[] rr = {"\u002b","\u003d"};
        for (int i = 0; i < 2; ++i) {
            GridBagConstraints c = new GridBagConstraints();
            c.fill = GridBagConstraints.BOTH;
            c.gridx = 3;
            c.gridy = 1 + 2*i;
            c.gridheight = 2;
            add(new JButton(rr[i]),c);
        }
          // zero
        GridBagConstraints czero = new GridBagConstraints();
        czero.fill = GridBagConstraints.BOTH;
        czero.gridx = 0;
        czero.gridy = 4;
        czero.gridwidth = 2;
        add(new JButton("0"),czero);
          // dot
        GridBagConstraints cdot = new GridBagConstraints();
        cdot.fill = GridBagConstraints.BOTH;
        cdot.gridx = 2;
        cdot.gridy = 4;
        cdot.weighty = 0.5;
        add(new JButton("\u2022"),cdot);
    }
}
```

9.10 Menus

Menus belong to the most useful elements of almost any graphical interface. In Java, one can easily create even quite complex menus. The classes involved are presented in the figure below



As we can see, the parent type here is **JMenuItem**, representing a selectable menu item. Selectable, because **JMenuItem** inherits from **JAbstractButton** and therefore can be clicked to fire an event that we can somehow handle — exactly as buttons. Also as buttons, menu items can be equipped with texts and icons that can be configured. In order to create a menu, one

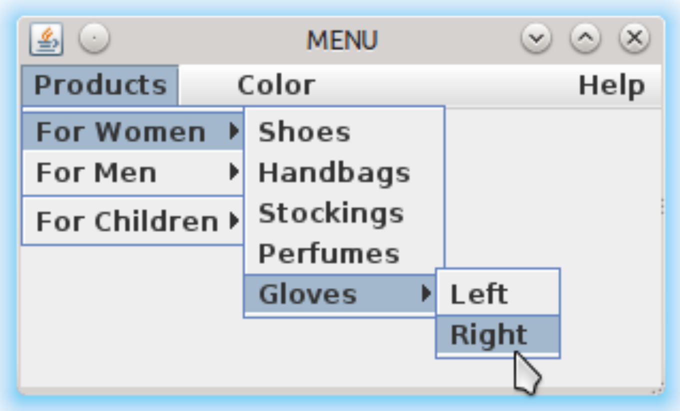
- creates **JMenuBar**;
- creates menus (**JMenu**);
- to each menu, adds other menus which will constitute its submenus;
- to each menu or submenu, adds other menus or, finally, menu items — **JMenuItem** — which therefore are leaves of the tree-like structures corresponding to each highest level menu;
- adds the highest level menus to **JMenuBar**;
- sets this menu bar as the menu bar of a window (usually **JFrame**).

Let us look at an example:

Listing 97 MDA-MenuSimple/MenuSimple.java

```java
import java.awt.Dimension;
import javax.swing.Box;
import javax.swing.JFrame;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JPanel;

public class MenuSimple {
    public static void main (String[] args) {
        JFrame f = new JFrame("MENU");
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JPanel panel = new JPanel();
        panel.setPreferredSize(new Dimension(300,130));

          // menu Products
        JMenu productMenu = new JMenu("Products");
          // submenu For Women
        JMenu women = new JMenu("For Women");
          // adding items to the submenu For Women
        women.add(new JMenuItem("Shoes"));
        women.add(new JMenuItem("Handbags"));
        women.add(new JMenuItem("Stockings"));
        women.add(new JMenuItem("Perfumes"));
        JMenu gloves = new JMenu("Gloves");
        gloves.add(new JMenuItem("Left"));
        gloves.add(new JMenuItem("Right"));
        women.add(gloves);
          // submenu For  Men
        JMenu men = new JMenu("For Men");
          // adding items to the submenu For Men
        men.add(new JMenuItem("Beer"));
          // submenu For Children (will be empty)
        JMenu children = new JMenu("For Children");
          // adding submenus to menu Products
        productMenu.add(women);
        productMenu.add(men);
        productMenu.addSeparator();
        productMenu.add(children);

          // menu Color
        JMenu colorMenu = new JMenu("Color");
          // adding menu items
        colorMenu.add(new JMenuItem("Red"));
        colorMenu.add(new JMenuItem("Blue"));

          // menu Help - here wil be empty...
        JMenu helpMenu  = new JMenu("Help");

          // adding menus to menu bar
        JMenuBar menuBar = new JMenuBar();
        menuBar.add(productMenu);
        menuBar.add(Box.createHorizontalStrut(20));
        menuBar.add(colorMenu);
        menuBar.add(Box.createGlue());
        menuBar.add(helpMenu);

          // setting menu bar
        f.setJMenuBar(menuBar);

        f.add(panel);
        f.pack();
        f.setLocationRelativeTo(null);
        f.setVisible(true);
    }
}
```

The program displays



Notice that

- By default, the view of the JMenuBar is laid out by Box layout. Therefore, one can add 'glues' and horizontal struts (rigid gaps) between menus: in the example above, there is a strut between 'Products' and 'Color' menus and a glue between 'Color' and 'Help' menus (and therefore 'Help' is shifted to the right).
- In the submenu lists, one can also add separators by invoking addSeparator; in the example there is such a separator between 'For Men' and 'For Children' submenus.

More examples are provided in section 10.2 (p. 173).

9.11 Dialogs

It is quite common that our application needs to read some data from the user, or display some kind of a message. This can be done by displaying a special type of a window — JDialog, which is somewhat similar to JFrame but with limited functionality. Objects of this class have an owner (parent window) that one can specify explicitly. The most convenient way of creating and using dialogs is by invoking one of the many static methods of class JOptionPane. These are:

- showConfirmDialog — asks the for confirmation (e.g., something like *'Do you really want to quit?'*); the possible answers are then 'yes', 'no', or 'cancel'.
- showInputDialog — prompts the user for some input data; returns a String typed by the user, or in some cases a user-selected Object.
- showMessageDialog — displays some kind of a message.
- showOptionDialog — combines functionality of the above three.

All these methods come in many overloaded flavors with different number of parameters. The meaning of these parameters is as follows:

- parentComponent — specifies the parent of the dialog (so it will appear on or just below its parent. Setting it to null will result in the dialog appearing in the center of the screen.

- message — a message to be placed in the dialog box (usually some kind of a prompt explaining the user what is expected). It does not to be a String — its interpretation depends on its type:
  - Object[] — an array of objects is interpreted as a series of messages arranged vertically.
  - Component — is displayed 'as is'.
  - Icon — the icon is displayed wrapped in a label.
  - for arguments of others types, toString is invoked and the returned string is displayed.
- messageType — defines the type of the message. Predefined types are specified by one of the integer constants from class JOptionPane: ERROR_MESSAGE, INFORMATION_MESSAGE, WARNING_MESSAGE, QUESTION_MESSAGE and PLAIN_MESSAGE. It will be taken into account for selecting, for example, an appropriate icon (if not set explicitly in the program).
- optionType — for dialogs displaying a set of buttons, specifies, if not set explicitly by the program, what buttons will appear. This is also an integer constants defined in class JOptionPane: DEFAULT_OPTION, YES_NO_OPTION, YES_NO_CANCEL_OPTION and OK_CANCEL_OPTION. However, by specifying explicitly options (see below), one can provide any set of buttons.
- options — specifies what buttons will appear in the dialog box: normally, this is an array of Strings, but can also be an array of any Objects. Then buttons will be created depending on the type of elements:
  - Component — the component will be used directly instead of a button;
  - Icon — the icon will be used instead of a button;
  - for other types, toString will be invoked and the returned string will be used on the button.
- icon — specifies a decorative icon to be placed in the dialog box. The default (if there is no corresponding argument or it is null) will be determined by the messageType parameter (see above).
- title — a title to appear on the title bar of the dialog.
- initialValue — the default selection (input value) if there are several options.

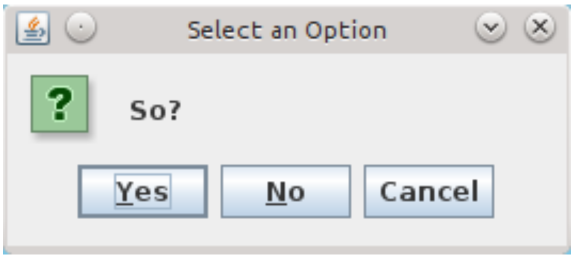
Some static methods of JOptionPane return an int; it is equal to one of the predefined constants from class JOptionPane: YES_OPTION, NO_OPTION, CANCEL_OPTION, OK_OPTION and CLOSED_OPTION and corresponds to a button that was clicked by the user (or, if the user just closed the dialog without pressing any button, CLOSED_OPTION is returned).

For example, when we want the user to confirm some decision, we can use a showConfirmDialog function. The following snippet

```
int a = JOptionPane.showConfirmDialog(
            null, // parent
            "So?" // message
        );
if (a == JOptionPane.YES_OPTION)
    System.out.println("Yes!");
else if (a == JOptionPane.NO_OPTION)
    System.out.println("No!");
```

```
else if (a == JOptionPane.CANCEL_OPTION)
    System.out.println("Canceled!");
else if (a == JOptionPane.CLOSED_OPTION)
    System.out.println("Closed!");
else
    System.out.println("Not possible...!");
```

will display a dialog



Select an Option

So?

and return an **int** which may then be examined, as shown in the example. Actually, there are four overloaded versions of **showConfirmDialog** (see documentation).

The **showConfirmDialog** function can give us only a simple yes/no answer. To request some data, **showInputDialog** may be used instead. It has six overloaded versions, five of which return a **String**. For example

```
String b = JOptionPane.showInputDialog(
               null, // parent
               "Enter an int...", // message
               "42" // initial value
           );
int i = 0;
if (b == null || b.trim().equals("")) {
    System.out.println("No value returned");
    i = -1;
} else {
    try {
        i = Integer.parseInt(b);
    } catch(NumberFormatException e) {
        System.out.println("This is not an int!");
        i = -2;
    }
}
System.out.println("i = " + i);
```

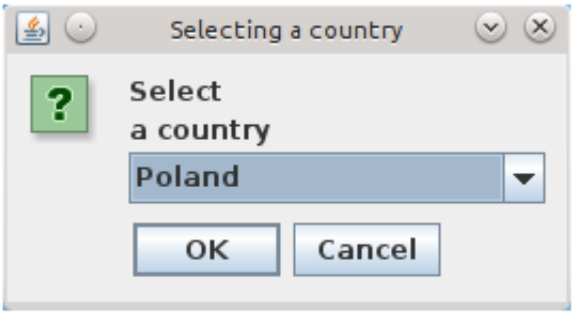
returns a **String** which can be then converted to an **int**. When such a dialog is displayed, the default value is already shown in the text field, so the user can just press ‘enter’ to select it.

There is one version of **showInputDialog** which returns an **Object**, not necessarily a **String**. Let us consider an example. Here we pass an array of references to object of type **CountryLabel** (which extends **JLabel**, but it could be any type). Options represented by the elements of the array, converted to strings by means of **toString** method, are shown in a combo box for the user to select from. After a selection has been made, the corresponding element of the array is returned as an **Object**, but can be safely cast to its real type:

Listing 98 MCW-Options/CountryDialog.java

```java
import java.awt.Dimension;
import javax.swing.ImageIcon;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JOptionPane;
import static javax.swing.JFrame.EXIT_ON_CLOSE;

public class CountryDialog {
    public static void main(String[] args) {
        JLabel[] opts = {
            new CountryLabel("Poland","Warsaw"),
            new CountryLabel("France","Paris"),
            new CountryLabel("UK","London")};
          // This version returns Object, not String!!!
          // Options will be displayed in a combo box.
        Object c = // will be of type CountryLabel
            JOptionPane.showInputDialog(
                null, // parent component
                "Select\na country",    // message
                "Selecting a country", // title
                JOptionPane.QUESTION_MESSAGE,// message type
                null,   // icon
                opts,   // options (objects of any type)
                opts[0] // default selection
            );
        CountryLabel cl = (CountryLabel)c;
        if (cl == null) System.out.println("Cancelled");
        else {
            JFrame f = new JFrame(cl.toString());
            f.setDefaultCloseOperation(EXIT_ON_CLOSE);
            f.add(cl);
            f.setSize(new Dimension(200,150));
            f.setLocationRelativeTo(null);
            f.setVisible(true);
        }
    }
}

class CountryLabel extends JLabel {
    String name;
    String capital;
    CountryLabel(String n, String c) {
        super(n + ", capital: " + c,
              new ImageIcon(n + ".gif"), JLabel.CENTER);
        setHorizontalTextPosition(JLabel.CENTER);
        setVerticalTextPosition(JLabel.BOTTOM);
        name=n;
        capital=c;
    }
    String getCountryName()  { return name; }
    String getCapital()      { return capital; }
    @Override
    public String toString() { return name; }
}
```

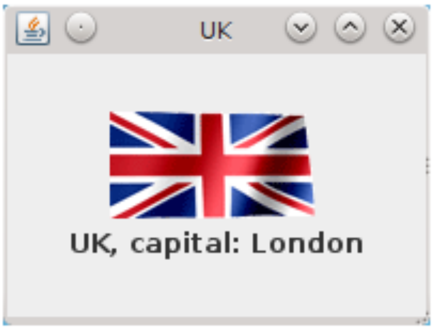
The program displays first a dialog with options in the form of a combo box, as shown on the left hand side of the figure below



Selecting a country

Select
a country

Poland



UK

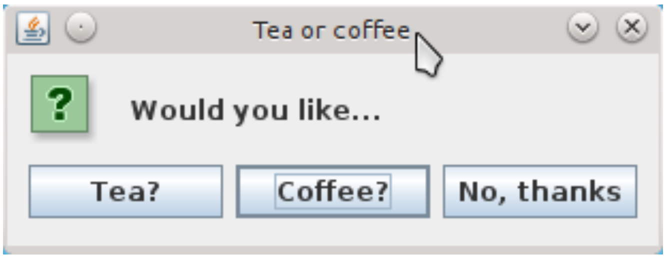
UK, capital: London

Note that pressing the space bar unfolds the options of the combo box; you can use ‘up’ and ‘down’ keys to move around the options, press ‘enter’ to select one and again ‘enter’ to finish the selection process. To jump directly to an option, you can just press the key corresponding to the first letter of an option. Therefore, you can do everything without even touching the mouse (power users abhor mice). In the example, after selecting ’UK’, you should see a little window shown on the right hand side of the figure.

Using **showOptionDialog**, one can customize texts on the buttons with options:

```java
Object[] opts = {"Tea?", "Coffee?", "No, thanks"};
int a = JOptionPane.showOptionDialog(
            null, // parent
            "Would you like...", // message
            "Tea or coffee",     // title
                // option type
            JOptionPane.YES_NO_CANCEL_OPTION,
                // message type
            JOptionPane.QUESTION_MESSAGE,
            null, // icon
            opts, // options
            opts[1] // initial value
        );
if (a == JOptionPane.YES_OPTION)
    System.out.println("Tea");
else if (a == JOptionPane.NO_OPTION)
    System.out.println("Coffee");
else if (a == JOptionPane.CANCEL_OPTION)
    System.out.println("Nothing");
else if (a == JOptionPane.CLOSED_OPTION)
    System.out.println("Closed!");
else
    System.out.println("Not possible...!");
```

For example, if option type is set to YES_NO_CANCEL_OPTION, instead of 'yes', 'no', 'cancel', we will see our texts passed as an array:



Would you like...

However, the returned value still will be one of YES_OPTION, NO_OPTION, CANCEL_OPTION or CLOSED_OPTION.

Another form of a dialog is presented below. Here we display a message dialog, but the function **showMessageDialog** doesn't return anything. However, the 'message' contains a group of radio buttons, and we can find which of them has been selected (after returning from the function) by querying this group of buttons:

Listing 99 MDE-RadioBut/RadioBut.java

```java
import java.util.Enumeration;
import javax.swing.AbstractButton;
import javax.swing.ButtonGroup;
import javax.swing.Icon;
import javax.swing.ImageIcon;
import javax.swing.JLabel;
import javax.swing.JFrame;
import javax.swing.JOptionPane;
import javax.swing.JRadioButton;

public class RadioBut {
    public static void main(String[] args) {
        Object[] mess =
            new Object[3+Stars.values().length];
        mess[0] = "Select one";
        mess[1] = "(and only one)";
        mess[2] = " ";
        ButtonGroup bgroup = new ButtonGroup();
        int i = 0;
        for (Stars s : Stars.values()) {
            JRadioButton b = new JRadioButton(s.getFirst());
            b.putClientProperty("star",s);
            mess[3+i] = b;
            bgroup.add(b);
            ++i;
        }

        JOptionPane.showMessageDialog(
                null,mess, // <-- array of Objects
                "Hard choice...",
                JOptionPane.QUESTION_MESSAGE, // ignored
                new ImageIcon("stars.png"));

        Stars star = null;
        Enumeration<AbstractButton> buttons =
                                    bgroup.getElements();
        while (buttons.hasMoreElements() && star == null) {
            AbstractButton b = buttons.nextElement();
            if (b.isSelected()) star =
                    (Stars)b.getClientProperty("star");
        }
        if (star != null) {
            JOptionPane.showMessageDialog(null,
                "You have selected " + star.getFirst() +
                " (" + star + ")", "Your selection",
                JOptionPane.INFORMATION_MESSAGE);
            display(star.getIcon());
        } else {
            JOptionPane.showMessageDialog(null,
                "No star selected!","No selection",
                JOptionPane.ERROR_MESSAGE);
        }
    }

    private static void display(Icon icon) {
        JFrame f = new JFrame("Your star");
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f.add(new JLabel(icon));
        f.pack();
        f.setLocationRelativeTo(null);
        f.setVisible(true);
    }
}

enum Stars {
    MONICA("monica") {
        @Override
        public String toString() {return "Monica Bellucci";}
    },
    PENELOPE("penelope") {
        @Override
        public String toString() {return "Penelope Cruz";}
    },
    CINDY("cindy") {
        @Override
        public String toString() {return "Cindy Crawford";}
    };
    Icon icon;
    Stars(String fname) {
        icon = new ImageIcon(fname + ".jpg");
    }
    Icon getIcon() { return icon; }
    String getFirst() {
        return this.toString().split(" ")[0];
    }
}
```

(the example also illustrates non-trivial use of enums, see sect. 3 on p. 41).

A few more examples of dialogs will be presented in sect. 10.2 on p. 173; in particular **JFileChooser** (see Listing 107, p. 189) and **JColorChooser** (see Listing 108, p. 192).

Section 10

GUI — events

10.1 General remarks

For a computer to be able to interact with the user, a special mechanism must be provided which handles **events**, like pressing a key on the keyboard or moving/dragging or clicking the mouse. As we know, Java has a specialized thread which handles such (and other) events, and, what is important, normally there is one and only one such thread, called **event-dispatch thread**. Events — key presses, mouse movements and clicks, etc. — are, of course, supplied by the operating system: the JVM intercepts them, wraps into objects of types derived from **EventObject** and enqueues them into the **event queue**. Event-dispatch thread dequeues them one by one in the same order as they were enqueued (so it is a FIFO queue) and passes them to *listeners* which has been registered as 'interested' in handling them (it is possible that one or more events will be coalesced into one). Most physical events will just be ignored, as there are no listeners interested in them.

Events are handled by call-back methods invoked on special objects which are registered as **listeners** of events originating from a given source (e.g., a graphical component, like a button) and of a specified type. Not all of them are purely 'physical', some are 'semantic' ones. For example events of type **ActionEvent** may be triggered by various physical events, like clicking a button, pressing 'enter' on the keyboard when a component of type **JTextField** has the focus, or pressing the space bar when the focus is on a **JButton**. But events may also be created without any physical cause — for example modifications of some data can trigger events in such a way that listeners interested in such events will be notified and can somehow react to them.

Let us recapitulate what we already know from the previous chapter: in order to handle events, we need

- a source of events; this can be a graphical component of our GUI or an object representing some data structure;
- a way to attach listeners of events of a specified type and taking place on a given source;
- listener — an object of a class implementing an appropriate interface and therefore providing definitions of its abstract methods; these methods will be invoked automatically as a reaction to an event.

To one source, we can attach many listeners, and the other way around: one listener can be attached to many sources. In order to delegate a listener as a handler of events, we usually invoke a special method

```
source.addXXXListener(listener);
```

where `XXX` specifies the type of events we are interested in; it can be, for example, `Action`, `Mouse`, `MouseMotion`, `Key` etc. The reference `source` refers to an object that has the ability to fire events of the specified type. As the argument, we pass an object which can handle events of the given type: its class has to implement a special interface which declares (as abstract methods) actions that are to be executed after an event of the given type occurred on the given source. Methods of the listener are called automatically: we never invoke them 'manually'. When an event on a given source occurs, the system checks if there are any objects delegated as listeners of this kind of events originating from this source and, if so, appropriate methods are invoked on these listeners. An object describing the event is passed as the argument. It has to succeed, because listeners must implement the corresponding interfaces (otherwise the program would not even compile). Event-handling methods are usually **public void** and they accept one argument: an object which carries information on the event as, for example, the source of the event, time of occurrence, and also other properties depending on the type of this particular event. For example, events of type **KeyEvent** (pressing a key on the keyboard) contain information about the key that has been pressed, whether or not the control or shift key was pressed simultaneously etc., while events of type **MouseEvent** carry information about the coordinates of the point where the mouse was located, whether the left, middle, or right button was pressed, etc.

There are many various types of events in Java, and therefore many interfaces that must be implemented by their listeners (only a few of them are presented below).

10.2 Action events

Action events are probably the simplest but at the same time very often used. These are 'semantic' events that can be created by various 'physical' actions. They can be fired, for example, by

- clicking a button
- pressing the space bar when a button has the focus;
- pressing 'enter' in an editing field;
- selecting an option from a menu;
- double-clicking on an item of a combo box;
- programmatically, by invoking the **doClick** method on a **JButon**.

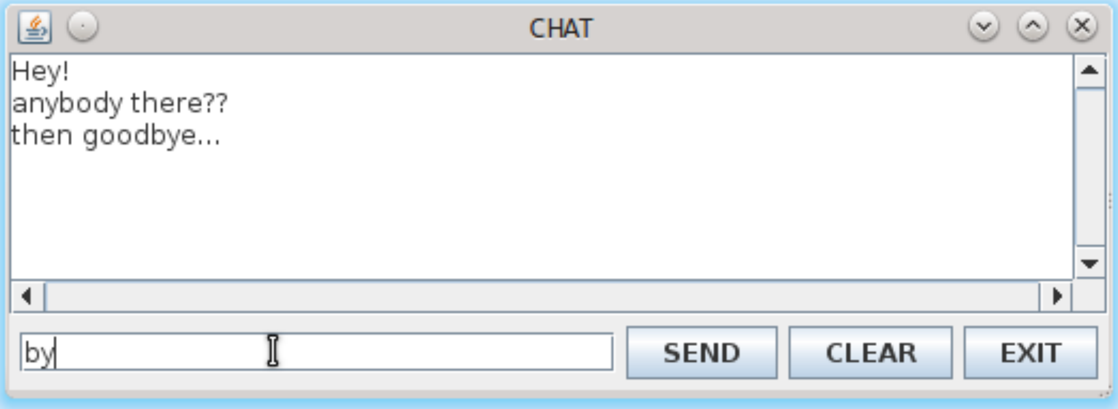
Listeners of action events have to implement the interface **ActionListener** which declares only one method

```
public void actionPerformed(ActionEvent e)
```

Declaring only one abstract method makes **ActionListener** a functional interface, which can be implemented by a lambda. An object listener implementing the interface can be delegated to play the rôle of a listener by invoking

```
source.addActionListener(listener);
```

Let us consider an example. The program below builds a simple GUI (resembling a typical chat GUI) with a text field, three buttons and a text area. Notice that the same action (variable send) is attached to both the text field tf and one of the buttons appearing in the southern panel.



Hey!
anybody there??
then goodbye...

by

Listener for the SEND action is defined as an object of anonymous class extending abstract class AbstractAction and providing implementation of the actionPerformed method. This is a convenient way to define a listener, because we can pass arguments to a constructor of AbstractAction, for example a text which should appear on the button and/or an icon. Of course, we can also use a separate class to define a listener; this is how the listener for bClear is created. As the class ClearListener is *not* an inner class, we have to pass references to our text field and text area (via the constructor), because the method actionPerformed operates on them. If the implementation of actionPerformed is sufficiently simple, we can use a lambda, as we did when creating a listener for the bExit button.

Listing 100 MCI-ChatGui/ChatGUI.java

```java
import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.AbstractAction;
import javax.swing.Action;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;
import javax.swing.SwingUtilities;

public class ChatGUI {

    public static void main(String[] args) {
        SwingUtilities.invokeLater(() -> createGUI());
    }
    private static void createGUI() {
        JFrame f = new JFrame("CHAT");
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JTextArea  ta = new JTextArea(7,20);
        JTextField tf = new JTextField(25);
          // object of an anonymous class
        Action send =
            new AbstractAction("SEND") {
                public void actionPerformed(ActionEvent e) {
                    ta.append(tf.getText()+'\n');
                    tf.setText("");
                    tf.requestFocus();
                }
            };
        tf.addActionListener(send);
        JScrollPane scr = new JScrollPane(ta,
                JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
                JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS);
        f.add(scr,BorderLayout.CENTER);

        JPanel south = new JPanel();
        south.add(tf);
        south.add(new JButton(send));
        JButton bClear = new JButton("CLEAR");
          // object of a separate class
        bClear.addActionListener(new ClearListener(ta,tf));
        JButton bExit = new JButton("EXIT");
          // lambda
        bExit.addActionListener(e -> System.exit(0));
        south.add(bClear);
        south.add(bExit);
        f.add(south,BorderLayout.SOUTH);

        f.pack();
        f.setLocationRelativeTo(null);
        f.setVisible(true);
        tf.requestFocus();
    }
}

class ClearListener implements ActionListener {
    JTextArea  ta;
    JTextField tf;
    ClearListener(JTextArea a, JTextField f) {
        ta = a;
        tf = f;
    }
    @Override
    public void actionPerformed(ActionEvent e) {
        ta.setText("");
        tf.setText("");
        tf.requestFocus();
    }
}
```

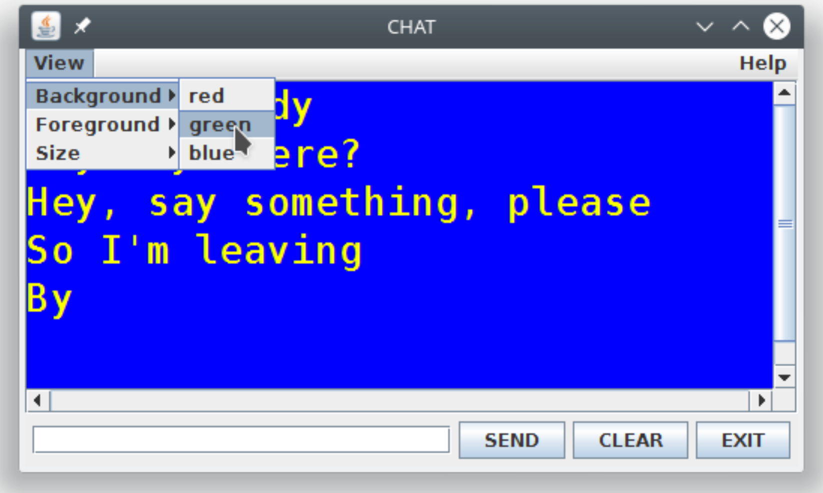
The next example is similar, but we added a menu (see sect. 9.10 on p. 161 and the example in Listing 97). Note that **JMenuItems** *are*, in a sense, buttons, as this class inherits from **AbstractButton**.

Listing 101 MCO-GuiMenu/ChatGUIMenu.java

```java
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.AbstractAction;
import javax.swing.Action;
import javax.swing.Box;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;
import javax.swing.SwingUtilities;

public class ChatGUIMenu {

    public static void main(String[] args) {
        final JFrame f = new JFrame("CHAT");
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        final JTextArea  ta = new JTextArea(7,20);
        final JTextField tf = new JTextField(25);
        Action send =
            new AbstractAction("SEND") {
                public void actionPerformed(ActionEvent e){
                    ta.append(tf.getText()+'\n');
                    tf.setText("");
                    tf.requestFocus();
                }
            };
        tf.addActionListener(send);
        JScrollPane scr = new JScrollPane(ta,
                JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
                JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS);
        f.add(scr,BorderLayout.CENTER);

        JPanel south = new JPanel();
        south.add(tf);
        south.add(new JButton(send));
        south.add(new JButton(
            new AbstractAction("CLEAR") {
                public void actionPerformed(ActionEvent e){
                    ta.setText("");
                    tf.setText("");
                    tf.requestFocus();
                }
            }
        ));
        south.add(new JButton(
            new AbstractAction("EXIT") {
                public void actionPerformed(ActionEvent e){
                    System.exit(0);
                }
            }
        ));

        f.add(south,BorderLayout.SOUTH);

        JMenuBar bar = new JMenuBar();
        JMenu mainMenu = new JMenu("View");
        JMenu helpMenu = new JMenu("Help");
        bar.add(mainMenu);
        bar.add(Box.createGlue());
        bar.add(helpMenu);
        f.setJMenuBar(bar);

        JMenu menuBG = new JMenu("Background");
        mainMenu.add(menuBG);
        ActionListener bgListener = new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                JMenuItem item = (JMenuItem)e.getSource();
                String s = item.getText();
                if        ("red".equals(s)) {
                    ta.setBackground(Color.RED);
                } else if ("green".equals(s)) {
                    ta.setBackground(Color.GREEN);
                } else if ("blue".equals(s)) {
                    ta.setBackground(Color.BLUE);
                }
            }
        };
        JMenuItem bgRed = new JMenuItem("red");
        bgRed.addActionListener(bgListener);
        JMenuItem bgGreen = new JMenuItem("green");
        bgGreen.addActionListener(bgListener);
        JMenuItem bgBlue = new JMenuItem("blue");
        bgBlue.addActionListener(bgListener);
        menuBG.add(bgRed);
        menuBG.add(bgGreen);
```

```java
        menuBG.add(bgBlue);

        JMenu menuFG = new JMenu("Foreground");
        mainMenu.add(menuFG);
        ActionListener fgListener = new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                JMenuItem item = (JMenuItem)e.getSource();
                String s = item.getText();
                if        ("white".equals(s)) {
                    ta.setForeground(Color.WHITE);
                } else if ("yellow".equals(s)) {
                    ta.setForeground(Color.YELLOW);
                } else if ("magenta".equals(s)) {
                    ta.setForeground(Color.MAGENTA);
                }
            }
        };
        JMenuItem fgWhite = new JMenuItem("white");
        fgWhite.addActionListener(fgListener);
        JMenuItem fgYellow = new JMenuItem("yellow");
        fgYellow.addActionListener(fgListener);
        JMenuItem fgMagenta = new JMenuItem("magenta");
        fgMagenta.addActionListener(fgListener);
        menuFG.add(fgWhite);
        menuFG.add(fgYellow);
        menuFG.add(fgMagenta);

        JMenu menuFN = new JMenu("Size");
        mainMenu.add(menuFN);
        ActionListener fnListener = new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                JMenuItem item = (JMenuItem)e.getSource();
                String s = item.getText();
                if        ("small".equals(s)) {
                    ta.setFont(new Font(
                            "Monospaced",Font.PLAIN,12));
                } else if ("medium".equals(s)) {
                    ta.setFont(new Font(
                            "Monospaced",Font.PLAIN,18));
                } else if ("big".equals(s)) {
                    ta.setFont(new Font(
                            "Monospaced",Font.PLAIN,26));
                }
            }
        };
        JMenuItem fnSmall = new JMenuItem("small");
        fnSmall.addActionListener(fnListener);
        JMenuItem fnMedium = new JMenuItem("medium");
        fnMedium.addActionListener(fnListener);
        JMenuItem fnBig = new JMenuItem("big");
        fnBig.addActionListener(fnListener);
        menuFN.add(fnSmall);
        menuFN.add(fnMedium);
        menuFN.add(fnBig);

        SwingUtilities.invokeLater(() -> {
            f.pack();
            f.setLocationRelativeTo(null);
            f.setVisible(true);
        });
    }
}
```

The program displays:



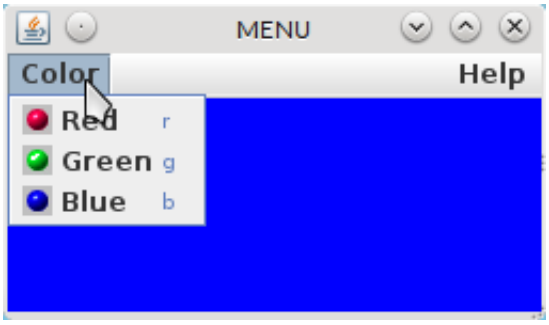
```
dy
ere?
Hey, say something, please
So I'm leaving
By
```

The example below shows how to create and handle menus and also how to attach a shortcut which can be used to select an option without even unfolding a menu (remember that JMenuItems are also a source of ActionEvents). It also shows how to use icons on menu items (which are in fact sort of buttons):

Listing 102 MCZ-Menu/Menu.java

```
import java.awt.Color;
import java.awt.Dimension;
import javax.swing.Box;
import javax.swing.ImageIcon;
import javax.swing.JFrame;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JPanel;
import javax.swing.KeyStroke;

public class Menu {
    public static void main (String[] args) {
        JFrame f = new JFrame("MENU");
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JPanel panel = new JPanel();
        panel.setPreferredSize(new Dimension(250,100));

          // main menu bar
        JMenuBar menuBar = new JMenuBar();

          // menu Color
        JMenu colorMenu = new JMenu("Color");
          // adding menu items`
        colorMenu.add(getItem(panel, "Red", Color.RED,
                    "red.gif",'r'));
        colorMenu.add(getItem(panel, "Green", Color.GREEN,
                    "green.gif",'g'));
        colorMenu.add(getItem(panel, "Blue", Color.BLUE,
                    "blue.gif",'b'));

          // menu Help
        JMenu helpMenu  = new JMenu("Help");

          // adding menus to menu bar
        menuBar.add(colorMenu);
        menuBar.add(Box.createGlue());
        menuBar.add(helpMenu);

          // setting menu bar
        f.setJMenuBar(menuBar);
        f.add(panel);
        f.pack();
        f.setLocationRelativeTo(null);
        f.setVisible(true);
    }
      // a helper static method
    private static JMenuItem getItem(JPanel p, String n,
                        Color c, String gif, char accel) {
        JMenuItem m = new JMenuItem(n,new ImageIcon(gif));
        m.setAccelerator(KeyStroke.getKeyStroke(accel));
        m.addActionListener(e -> {
            p.setBackground(c); p.repaint();
        });
        return m;
    }
}
```

which shows



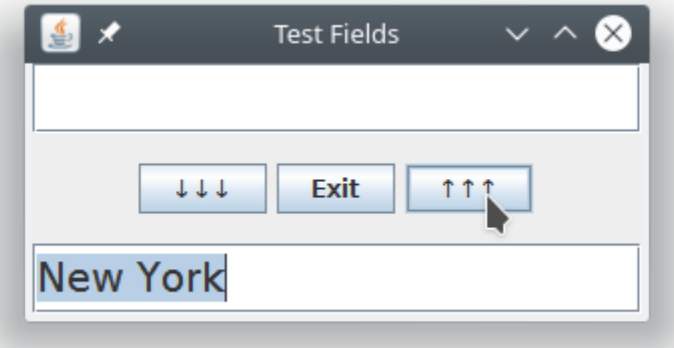
The next example also illustrates handling **ActionEvents**s:

Listing 103 MCR-TextFields/TextFields.java

```java
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.Font;
import java.awt.FlowLayout;
import java.awt.GridLayout;
import javax.swing.AbstractAction;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JTextField;
import javax.swing.SwingUtilities;

public class TextFields extends JFrame {

    public static void main(String[] args) {
        new TextFields();
    }

    TextFields() {
        super("Test Fields");
        setDefaultCloseOperation(EXIT_ON_CLOSE);

        setLayout(new GridLayout(3,1,5,10));

        JTextField lower = new JTextField(15);
        JTextField upper = new JTextField(15);
        lower.setFont(new Font("Dialog",Font.PLAIN,20));
        upper.setFont(new Font("Dialog",Font.PLAIN,20));

          // listener as an object of an anonymous class
        lower.addActionListener(
            new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    upper.requestFocus();
                    upper.selectAll();
                }
            });
          // listener as na object of an external class
        upper.addActionListener(new UpperListener(lower));

        JButton u2l  = new JButton(
            new AbstractAction("\u2193\u2193\u2193") {
                public void actionPerformed(ActionEvent e) {
                    lower.setText(upper.getText());
                    upper.setText("");
                    lower.selectAll();
                    lower.requestFocus();
                }
            });
        JButton exit = new JButton(
            new AbstractAction("Exit") {
                public void actionPerformed(ActionEvent e) {
                    System.exit(0);
                }
            });
        JButton l2u  = new JButton(
            new AbstractAction("\u2191\u2191\u2191") {
                public void actionPerformed(ActionEvent e) {
                    upper.setText(lower.getText());
                    lower.setText("");
                    upper.selectAll();
                    upper.requestFocus();
                }
            });
        JPanel center = new JPanel();
        center.setLayout(new FlowLayout());
        center.add(u2l); center.add(exit); center.add(l2u);

        add(upper);
        add(center);
        add(lower);

        SwingUtilities.invokeLater(() -> {
            pack();
            setLocationRelativeTo(null);
            setVisible(true);
        });
    }
}

class UpperListener implements ActionListener {
    JTextField tf;

    UpperListener(JTextField tf) {
        this.tf = tf;
    }
```

```java

    public void actionPerformed(ActionEvent e) {
        tf.requestFocus();
        tf.selectAll();
    }
}
```

The program displays



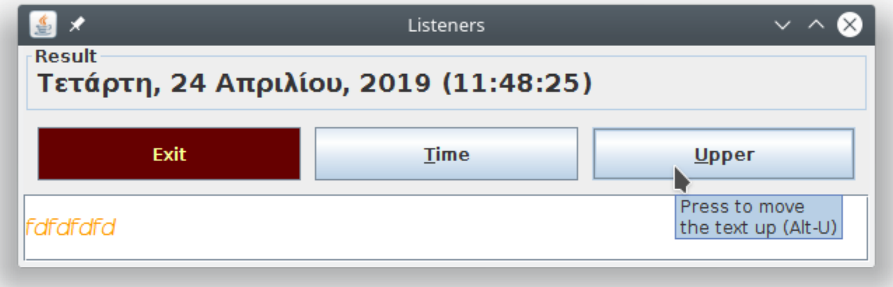
New York

Components which fire action events are often equipped with tool tips and mnemonics — this is shown in the example below:

Listing 104 MCV-Listeners/Listeners.java

```java
import java.awt.Color;
import java.awt.GridLayout;
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.Date;
import java.util.Locale;
import java.text.SimpleDateFormat;
import javax.swing.BorderFactory;
import javax.swing.JButton;
import javax.swing.JLabel;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JTextField;
import javax.swing.SwingUtilities;
import static java.awt.event.KeyEvent.VK_T;
import static java.awt.event.KeyEvent.VK_U;

public class Listeners extends JFrame {

    public static void main(String[] args) {
        new Listeners("Listeners");
    }

    Listeners(String windowTitle) {
        super(windowTitle);
        setDefaultCloseOperation(EXIT_ON_CLOSE);

          // Label
        JLabel label = new JLabel();
        label.setFont(new Font("SansSerif",Font.BOLD,18));
        label.setBorder(BorderFactory.createCompoundBorder(
            BorderFactory.createTitledBorder("Result"),
            BorderFactory.createEmptyBorder(10,5,16,5)));

          // Text
        JTextField field = new JTextField(40);
        field.setFont(new Font("Dialog",Font.ITALIC,16));
        field.setForeground(new Color(255,153,0));

          // First button - Exit
        JButton bExit = new JButton("Exit");
        bExit.setBackground(new Color(102,0,0));
        bExit.setForeground(new Color(255,255,153));
        bExit.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                System.exit(0);
            }
        });

          // Second button - Time
        JButton bTime = new JButton("Time");
        bTime.setToolTipText("<html>Press to see<br />" +
                             "current time (Alt-T)");
        bTime.setMnemonic(VK_T);
        bTime.addActionListener(
                new TimeListener(label,field));

          // Third button - convert to upper case.
          // Same action on text field
        JButton bUppe = new JButton("Upper");
        bUppe.setToolTipText("<html>Press to move<br />" +
                             "the text up (Alt-U)");
        bUppe.setMnemonic(VK_U);
        ActionListener act = new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                String s = field.getText().toUpperCase();
                label.setText(s);
                field.setText("");
                field.requestFocus();
            }
        };
        bUppe.addActionListener(act);
        field.addActionListener(act);

          // panel with buttons
        JPanel buttons =
                new JPanel(new GridLayout(1,0,10,2));
        buttons.setBorder(
                BorderFactory.createEmptyBorder(5,9,5,9));
        buttons.add(bExit);
        buttons.add(bTime);
        buttons.add(bUppe);

          // layout of the frame (its ContentPane)
        setLayout(new GridLayout(0,1,10,5));

        add(label);
        add(buttons);
        add(field);

        SwingUtilities.invokeLater(() -> {
            pack();
            setLocationRelativeTo(null);
            setVisible(true);
            field.requestFocus();
        });
    }
}

class TimeListener implements ActionListener {
    JLabel           label;
    JTextField       field;
    SimpleDateFormat dateFormat;

    TimeListener(JLabel label, JTextField field) {
        this.label = label;
        this.field = field;
        dateFormat = new SimpleDateFormat(
            "EEEE, d MMMM, yyyy (kk:mm:ss)",
            new Locale("el","GR"));
    }

    public void actionPerformed(ActionEvent e) {
        String s = dateFormat.format(new Date());
        label.setText(s);
        field.requestFocus();
    }
}
```

which displays



Τετάρτη, 24 Απριλίου, 2019 (11:48:25)

Press to move the text up (Alt-U)

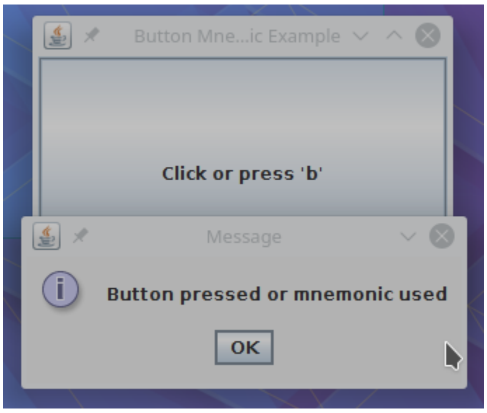
fdfdfdfd

Mnemonics (and tool tips) may also be set by explicit use of input and action maps:

Listing 105 MES-InpActMaps/ButtAct.java

```java
import java.awt.Dimension;
import java.awt.event.ActionEvent;
import javax.swing.AbstractAction;
import javax.swing.Action;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JOptionPane;
import javax.swing.KeyStroke;

public class ButtAct {
    public static void main(String[] args) {
        JFrame f = new JFrame("Button Mnemonic Example");
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        Action bAction = new BAction("Click or press 'b'",
                                     "just a button...");
        JButton button = new JButton(bAction);
        button.getInputMap()
              .put(KeyStroke.getKeyStroke('b'),
                   "Click Me Button");
        button.getActionMap()
              .put("Click Me Button", bAction);
        f.add(button);
        f.setSize(new Dimension(300,200));
        f.setLocationRelativeTo(null);
        f.setVisible(true);
    }
}

class BAction extends AbstractAction {
    public BAction(String text, String d) {
        super(text);
        putValue(SHORT_DESCRIPTION, d);
    }
    @Override
    public void actionPerformed(ActionEvent e) {
        JOptionPane.showMessageDialog(null,
            "Button pressed or mnemonic used");
    }
}
```

The program displays



Click or press 'b'

Button pressed or mnemonic used

Very often, we can attach the same listener to many sources of common type, e.g., **JButton**. When events arrive to the listener's method, we have to know which component has fired the event and, possibly, we need some additional information from the source. Fortunately, there is a simple mechanism by which we can pass such additional information from the source of na event to its receiver. For sources extending **AbstractButton** (as **JButton**, **JMenuItem** or **JToggleButton**) we can, by invoking **setActionCommand(String)**, assign a string to the component, which then can be read from the source of the event (`event.getSource()`) by invoking **getActionCommand**. If not set explicitly, action command is just a label on the button. This, however, is not always practical, as many buttons may have the same text on them, or this text depends, e.g., on localization, or it is modified dynamically during execution. If we set the action command explicitly, we can ensure its uniqueness and its constant value.

Even more powerful mechanism for passing information from the source of an event to its receiver is provided by the so called client properties. Any **JComponent** contains a, initially empty, map with both keys and values declared as having type **Object**. Calling on any component

`putClientProperty(key,value)`

we can add any entry to this map. The values put to the map can then be fetched by invoking on the source of an event

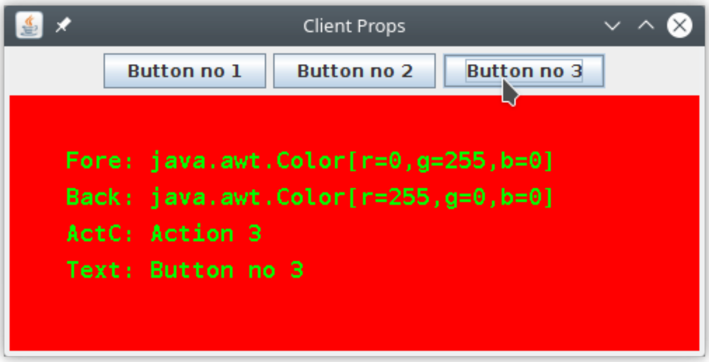
`getClientProperty(key)`

Below, we present an example of both mechanism:

Listing 106 MET-CliProp/CliProp.java

```java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class CliProp extends JFrame
                     implements ActionListener {
    MyPanel center = new MyPanel();

    public static void main(String[] args) {
        new CliProp();
    }
    private CliProp() {
        super("Client Props");
        setDefaultCloseOperation(DISPOSE_ON_CLOSE);
        center.setFont(new Font("Monospaced",Font.BOLD,16));
        Color[]  fcols = {Color.RED,Color.BLUE,Color.GREEN};
        Color[]  bcols = {Color.BLUE,Color.BLACK,Color.RED};
        JPanel north = new JPanel();

        for (int i = 0; i < fcols.length; ++i) {
            JButton b = new JButton("Button no " +(i+1));
            b.putClientProperty("fcolor",fcols[i]);
            b.putClientProperty("bcolor",bcols[i]);
            b.setActionCommand("Action " + (i+1));
            b.addActionListener(this);
            north.add(b);
        }
        add(north, BorderLayout.NORTH);
        center.setPreferredSize(new Dimension(300,200));
        add(center, BorderLayout.CENTER);

        setSize(new Dimension(500,250));
        setLocationRelativeTo(null);
        setVisible(true);
    }

    public void actionPerformed(ActionEvent e) {
        JButton but = (JButton)e.getSource();
        Color f = (Color)but.getClientProperty("fcolor");
        Color b = (Color)but.getClientProperty("bcolor");
        center.setMessages(new String[]{
                "Fore: " + f,
                "Back: " + b,
                "ActC: " + but.getActionCommand(),
                "Text: " + but.getText()
        });
        center.setForeground(f);
        center.setBackground(b);
        center.repaint();
    }

    class MyPanel extends JPanel {
        String[] mesg = {""};
        void setMessages(String[] m) { mesg = m; }
        @Override
        public void paintComponent(Graphics g) {
            super.paintComponent(g);
            int h = getHeight();
            int step = h/(mesg.length+3), y = 2*step;
            for (int i=0; i < mesg.length; ++i, y += step)
                g.drawString(mesg[i], 40, y);
        }
    }
}
```

The program displays



```
Fore: java.awt.Color[r=0,g=255,b=0]
Back: java.awt.Color[r=255,g=0,b=0]
ActC: Action 3
Text: Button no 3
```

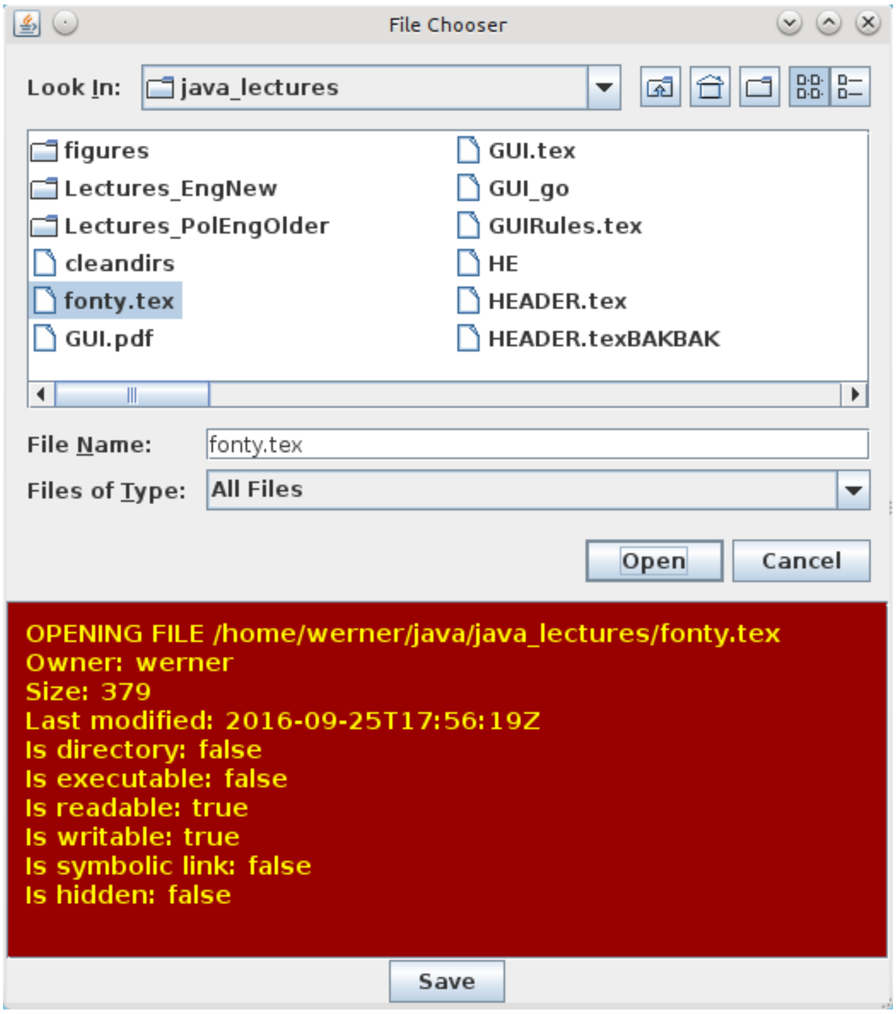
Action listeners can also be attached to dialogs of type **JFileChooser**. Its static method **showOpenDialog** displays a modal dialog and returns an **int**, and if it's equal to the constant APPROVE_OPTION defined in **JFileChooser** class, then we can fetch the file selected by invoking **getSelectedFile** on a **JFileChooser**. This, however, will not work if we embed the dialog into a component and keep it open. In such situation, we can attach an action listener to the **JFileChooser** component; an event passed to **actionPerformed** will carry, in its 'action command', information about whether a file has been selected (then it will be equal to string APPROVE_SELECTION from class **JFileChooser**). Both approaches are illustrated in the following example:

Listing 107 MDF-FChooser/FChooser.java

```
import java.awt.Color;
import java.awt.Font;
import java.awt.Insets;
import java.awt.event.ActionEvent;
import java.io.File;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import javax.swing.BoxLayout;
import javax.swing.JButton;
```

```java
import javax.swing.JFileChooser;
import javax.swing.JFrame;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import static java.awt.Component.CENTER_ALIGNMENT;

public class FChooser extends JPanel {
    public FChooser() {
        JOptionPane.showMessageDialog(null,
                "First select a directory...",
                "Message",
                JOptionPane.INFORMATION_MESSAGE);
        JFileChooser dir = new JFileChooser(
                    System.getProperty("user.dir"));
        dir.setFileSelectionMode(
                    JFileChooser.DIRECTORIES_ONLY);
        File root = null;
        while (root == null) {
            int r = dir.showOpenDialog(null);
            if (r == JFileChooser.APPROVE_OPTION)
                root = dir.getSelectedFile();
            else if (r == JFileChooser.CANCEL_OPTION)
                System.exit(1);
        }

        JFileChooser fc = new JFileChooser(root);
        fc.setFileSelectionMode(
                JFileChooser.FILES_AND_DIRECTORIES);
        JTextArea ta = new JTextArea(11,30);
        fc.addActionListener(e -> {
            if (e.getActionCommand().equals(
                  JFileChooser.APPROVE_SELECTION))
                info(fc.getSelectedFile().toPath(),ta,
                     "OPENING FILE");
        });

        ta.setMargin(new Insets(10,10,10,10));
        ta.setEditable(false);
        ta.setFont(new Font("Dialog",Font.BOLD,14));
        ta.setBackground(new Color(153,0,0));
        ta.setForeground(new Color(255,255,0));
        JScrollPane scroll = new JScrollPane(ta);

        JButton save = new JButton("Save");
        save.setAlignmentX(CENTER_ALIGNMENT);
        save.addActionListener(e -> {
            JFileChooser sav = new JFileChooser(
                    System.getProperty("user.home"));
            int r = sav.showSaveDialog(ta);
            if (r == JFileChooser.APPROVE_OPTION)
                info(sav.getSelectedFile().toPath(),ta,
                     "SAVING FILE");
        });
        setLayout(new BoxLayout(this,BoxLayout.Y_AXIS));
        add(fc);
        add(scroll);
        add(save);
    }

    private void info(Path path, JTextArea ta, String msg) {
        ta.setText("");
        String nl = System.getProperty("line.separator");
        StringBuilder sb = new StringBuilder(200);
        sb.append(msg + " " + path + nl);
        boolean exists = Files.exists(path);
        if (!exists)
            sb.append("NEW FILE");
        else {
            try {
                sb.append("Owner: " +
                    Files.getOwner(path) + nl);
                sb.append("Size: " +
                    Files.size(path) + nl);
                sb.append("Last modified: " +
                    Files.getLastModifiedTime(path) + nl);
                sb.append("Is directory: " +
                    Files.isDirectory(path) + nl);
                sb.append("Is executable: " +
                    Files.isExecutable(path) + nl);
                sb.append("Is readable: " +
                    Files.isReadable(path) + nl);
                sb.append("Is writable: " +
                    Files.isWritable(path) + nl);
                sb.append("Is symbolic link: " +
                    Files.isSymbolicLink(path) + nl);
                sb.append("Is hidden: " +
                    Files.isHidden(path) + nl);
            } catch(IOException e) {
                sb.append("ERROR WHEN RETRIEVING INFO");
            }
        }
        ta.setText(sb.toString());
    }

    public static void main(String[] args) {
        JFrame f = new JFrame("File Chooser");
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f.add(new FChooser());
        f.pack();
        f.setLocationRelativeTo(null);
        f.setVisible(true);
    }
  }
```

which shows



Look In: java_lectures

File Name: fonty.tex

Files of Type: All Files

```
OPENING FILE /home/werner/java/java_lectures/fonty.tex
Owner: werner
Size: 379
Last modified: 2016-09-25T17:56:19Z
Is directory: false
Is executable: false
Is readable: true
Is writable: true
Is symbolic link: false
Is hidden: false
```

A similar approach may be used in the case of color selection dialog (JColorChooser). This time, we do not attach an action listener, but rather ChangeListener, but this is also a very simple functional interface with only one abstract method stateChanged. However, it has to be attached *not* to the component JColorChooser itself, but to its **model**, available by invoking getSelectionModel returning ColorSelectionModel (more on models, later).

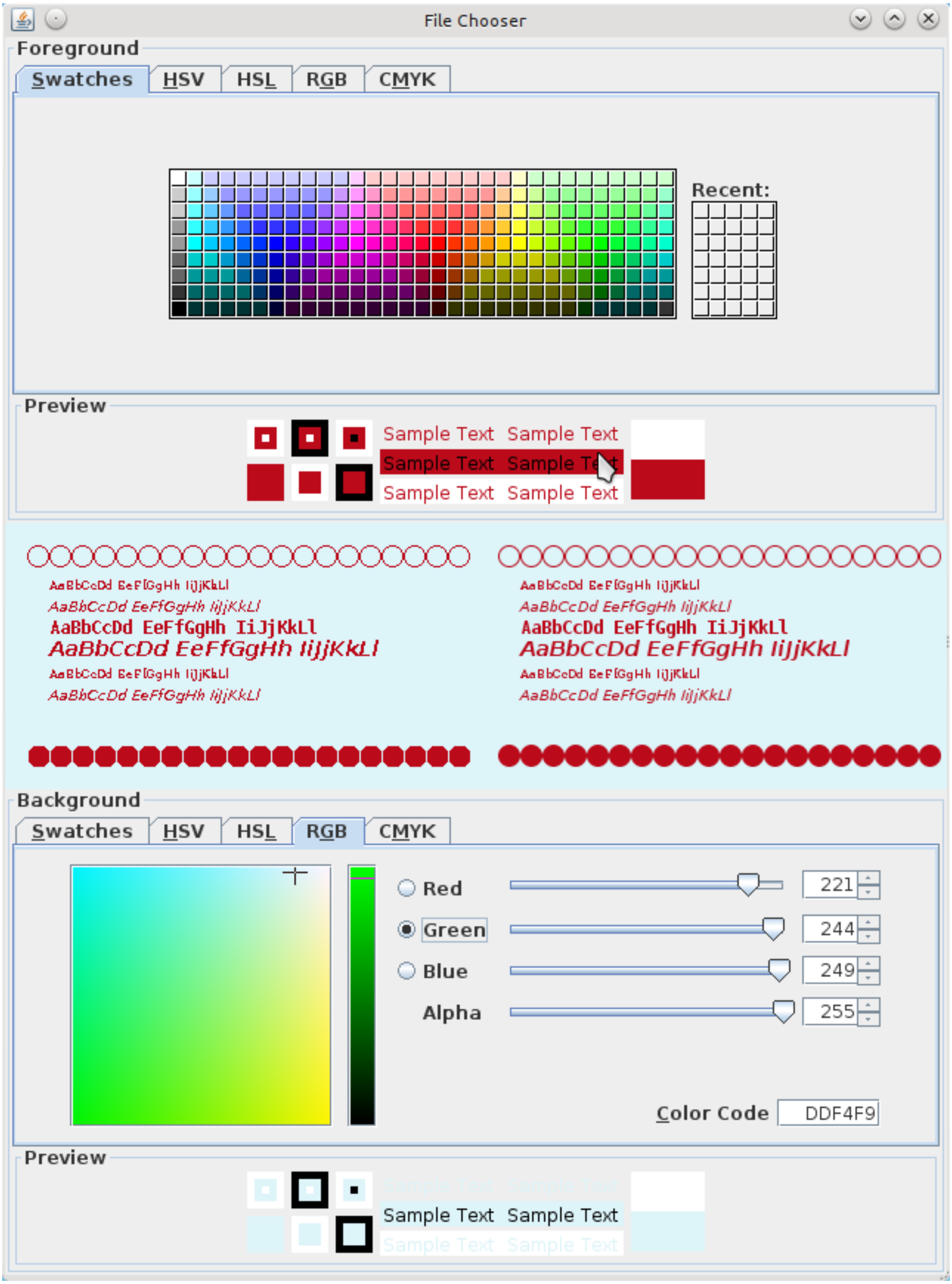
The example below shows an application which allows the user to experiment with foreground and background colors:

Listing 108 MDG-CChooser/CChooser.java

```
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.RenderingHints;
import javax.swing.BorderFactory;
import javax.swing.BoxLayout;
import javax.swing.JColorChooser;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.colorchooser.ColorSelectionModel;

public class CChooser extends JPanel {
  public CChooser() {
    MyPanel panel = new MyPanel(Color.YELLOW,Color.BLUE);
    panel.setPreferredSize(new Dimension(300,180));

    JColorChooser fg = new JColorChooser();
    fg.setBorder(BorderFactory.createTitledBorder(
                    "Foreground"));
    ColorSelectionModel modelFG = fg.getSelectionModel();
     // lambda defines stateChanged(ChangeEvent)
     // in ChangeListener functional interface
    modelFG.addChangeListener(e -> {
      panel.setFG(modelFG.getSelectedColor());
    });

    JColorChooser bg = new JColorChooser();
    bg.setBorder(BorderFactory.createTitledBorder(
                    "Background"));
    ColorSelectionModel modelBG = bg.getSelectionModel();
     // lambda defines stateChanged(ChangeEvent)
     // in ChangeListener functional interface
    modelBG.addChangeListener(e -> {
      panel.setBG(modelBG.getSelectedColor());
    });

    setLayout(new BoxLayout(this,BoxLayout.Y_AXIS));
    add(fg);
    add(panel);
    add(bg);
  }

  public static void main(String[] args) {
    JFrame f = new JFrame("File Chooser");
    f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    f.add(new JScrollPane(new CChooser(),
         JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
         JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS));
    f.pack();
    f.setLocationRelativeTo(null);
    f.setVisible(true);
```

```java
    }
}

class MyPanel extends JPanel {
    Color fg, bg;
    private static String st = "AaBbCcDd EeFfGgHh IiJjKkLl";
    private static Font[] fonts = {
        new Font("Serif", Font.PLAIN, 8),
        new Font("SansSerif", Font.ITALIC, 10),
        new Font("Monospaced", Font.BOLD, 12),
        new Font("Dialog", Font.ITALIC | Font.BOLD, 14),
    };
    MyPanel(Color fg, Color bg) {
        this.fg = fg;
        this.bg = bg;
    }
    @Override
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        setBackground(bg);
        g.setColor(fg);
        int h = getHeight(), w = getWidth(),
            s = 15, half = w/2;
          // no antialiasing
        for (int x=s, y1=s, y2=h-2*s; x < half-s; x += s) {
            g.drawOval(x,y1,s-1,s-1);
            g.fillOval(x,y2,s,s);
        }
        int y = 3*s;
        int i = 0;
        while (y < h-3*s) {
            g.setFont(fonts[i]);
            g.drawString(st,2*s,y);
            y += s;
            i = (i+1)%fonts.length;
        }

        setAntialiasing(g);

          // with antialiasing
        for (int x=half+s, y1=s, y2=h-2*s; x < w-s; x += s) {
            g.drawOval(x,y1,s-1,s-1);
            g.fillOval(x,y2,s,s);
        }
        y = 3*s;
        i = 0;
        while (y < h-3*s) {
            g.setFont(fonts[i]);
            g.drawString(st,half+2*s,y);
            y += s;
            i = (i+1)%fonts.length;
        }
    }
    public void setFG(Color c) {
        fg = c;
        repaint();
    }
    public void setBG(Color c) {
        bg = c;
        setBackground(c);
        repaint();
    }
    private static void setAntialiasing(Graphics g) {
        Graphics2D g2 = (Graphics2D)g;
        g2.setRenderingHint(
            RenderingHints.KEY_ANTIALIASING,
            RenderingHints.VALUE_ANTIALIAS_ON);
        g2.setRenderingHint(
            RenderingHints.KEY_TEXT_ANTIALIASING,
            RenderingHints.VALUE_TEXT_ANTIALIAS_ON);
    }
}
```

(the example shows also the influence of switching anti-aliasing on and off). The interface displayed by the program looks like this:



○ Red 221

● Green 244

○ Blue 249

Alpha 255

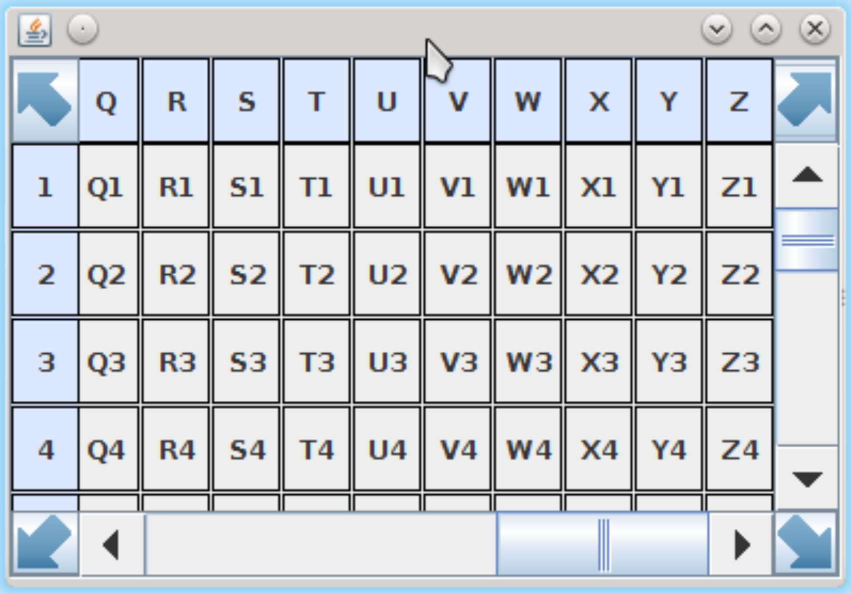
Color Code DDF4F9

Yet another example of using action events to control the graphical interface:

Listing 109 MCX-ScrollView/ScrollEx.java

```java
import java.awt.Color;
import java.awt.Dimension;
import java.awt.GridLayout;
import java.awt.Rectangle;
import javax.swing.ImageIcon;
import javax.swing.BorderFactory;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.UIManager;

public class ScrollEx extends JFrame {
    public static void main(String[] args) {
        new ScrollEx();
    }

    ScrollEx() {
        setDefaultCloseOperation(EXIT_ON_CLOSE);

          // colors
        Color heaC = new Color(219,232,255),
              fraC = Color.BLACK;
          // cell dimensions
        Dimension cellDim = new Dimension(32,40),
                  cellRow = new Dimension(32,40);
        int rows = 30, cols = 26;
        String[] lit = new String[cols];

          // row of column headers
        JPanel colHead = new JPanel(
                new GridLayout(1,cols,1,1));
        colHead.setBackground(heaC);
        for (int c = 0; c < cols; ++c) {
            lit[c] = "" + (char) ('A'+c);
            colHead.add(createLabel(lit[c],fraC,cellDim));
        }

          // column of row headers
        JPanel rowHead = new JPanel(
                new GridLayout(rows,1,1,1));
        rowHead.setBackground(heaC);
        for (int r = 0; r < rows; ++r)
            rowHead.add(createLabel(""+(r+1),fraC,cellRow));

          // content
        JPanel content = new JPanel(
                new GridLayout(rows, cols,1,1));
        for (int r = 0; r < rows; ++r)
            for (int c = 0; c < cols; ++c)
                content.add(createLabel(lit[c]+(r+1),
                                 fraC, cellDim));

        UIManager.put("ScrollBar.width", 32);
        JScrollPane scroll = new JScrollPane();
        scroll.setViewportView(content);
        scroll.setRowHeaderView(rowHead);
        scroll.setColumnHeaderView(colHead);

          // corners
        JButton upperLeft = new JButton(
                    new ImageIcon("ArrUL.png"));
        JButton lowerLeft = new JButton(
                    new ImageIcon("ArrLL.png"));
        JButton upperRight = new JButton(
                    new ImageIcon("ArrUR.png"));
        JButton lowerRight = new JButton(
                    new ImageIcon("ArrLR.png"));
        upperLeft.addActionListener(e -> {
            content.scrollRectToVisible(
                    new Rectangle(0,0,1,1));
        });
        lowerLeft.addActionListener(e -> {
            content.scrollRectToVisible(
                    new Rectangle(0,
                            content.getHeight()-1,1,1));
        });
        upperRight.addActionListener(e -> {
            content.scrollRectToVisible(
                    new Rectangle(content.getWidth()-1,
                            0,1,1));
        });
        lowerRight.addActionListener(e -> {
            content.scrollRectToVisible(
                    new Rectangle(content.getWidth()-1,
                           content.getHeight()-1,1,1));
        });
        scroll.setCorner(JScrollPane.UPPER_LEFT_CORNER,
                                             upperLeft);
        scroll.setCorner(JScrollPane.UPPER_RIGHT_CORNER,
                                             upperRight);
        scroll.setCorner(JScrollPane.LOWER_LEFT_CORNER,
                                             lowerLeft);
        scroll.setCorner(JScrollPane.LOWER_RIGHT_CORNER,
                                             lowerRight);

          // the whole grid will not fit inside...
        add(scroll);
        setPreferredSize(new Dimension(750,550));
        pack();
        setLocationRelativeTo(null);
        setVisible(true);
    }

    JLabel createLabel(String s, Color fC, Dimension dim) {
        JLabel lab = new JLabel(s, JLabel.CENTER);
        lab.setBorder(BorderFactory.createLineBorder(fC));
        lab.setPreferredSize(dim);
        return lab;
    }
}
```

which displays



	Q	R	S	T	U	V	W	X	Y	Z
1	Q1	R1	S1	T1	U1	V1	W1	X1	Y1	Z1
2	Q2	R2	S2	T2	U2	V2	W2	X2	Y2	Z2
3	Q3	R3	S3	T3	U3	V3	W3	X3	Y3	Z3
4	Q4	R4	S4	T4	U4	V4	W4	X4	Y4	Z4

10.3 Mouse events

The mouse is a source of ‘physical’ events of type MouseEvent. These can be handled by listeners implementing two different interfaces. Both are *not* functional interfaces, as they declare more than one abstract method.

The first one is MouseListener, which has five methods:

```
public interface MouseListener extends EventListener {
    public void mousePressed(MouseEvent e);
    public void mouseReleased(MouseEvent e);
    public void mouseClicked(MouseEvent e);
    public void mouseEntered(MouseEvent e);
```

```
        public void mouseExited(MouseEvent e);
    }
```

These functions will be called

- `mousePressed` — when any button of the mouse is pressed. The event passed as argument will contain information about which button (left, middle, or right) it was, whether ***Shift***, ***Ctrl*** or ***Alt*** buttons were also pressed simultaneously, the coordinates of the point at which the mouse cursor was at the moment of a click, etc;
- `mouseReleased` — when any button of the mouse is released. The event passed as argument will of course contain information about the context of the event;
- `mouseClicked` — when any button of the mouse is clicked, i.e., pressing and releasing the button happened at the same location and within sufficiently small time interval;
- `mouseEnterd` — when the mouse enters the area of the component which is the source of the event listened to (no button has to be pressed);
- `mouseExited` — when the mouse leaves the area of the component which is the source of the events listened to (no button has to be pressed).

The other interface dealing with mouse events is **MouseMotionListener** (notice that the type of the corresponding events is the same as before, **MouseEvent** — there is *no* type **MouseMotionEvent**). The interface declares two methods:

```
    public interface MouseMotionListener extends EventListener {
        public void mouseDragged(MouseEvent e);
        public void mouseMoved(MouseEvent e);
    }
```

Events of type **MouseEvent** are fired in very quick succession whenever we move the mouse: if any mouse button *is* pressed, then **mouseDragged** will be invoked with small time interval between events; if mouse is moved without pressing any button, **mouseMoved** will be invoked. Remember, that events are fired with very high frequency so there are thousands of them. This means that creating the events and inserting them into the event queue may be very expensive. Therefore, attaching listeners of these type of events is something that should be avoided if not necessary. This is also the reason why handling these mouse events has been assigned a special type of listeners; if we do not need to react to the moves of the mouse (only pressing its buttons), we do not need to attach any **MouseMotionListener** and all these events may be safely ignored by the JVM.

There is also an interface **MouseInputListener** (but *no* the corresponding **addMouseInputListener** method!) which combines both **MouseListener** and **MouseMotionListener** (seven methods to implement!).
Quite often, we are only interested in one type of mouse events, e.g. only in clicks. Still, to implement the interface, we have to override all five (or seven) methods (with empty implementation of those which are of no interest for us). However, if our class does not extend any class (except the default **Object**), we can make it extend the concrete class **MouseAdapter** — this class implements all seven methods of **MouseInputListener** and provides empty implementation of all of them. If our class extends **MouseAdapter**, we have to implement only those methods which we need, all other already have empty implementation.

Let us look at the example showing all mouse events in action. Here, class **MListener** implements **MouseInputListener** and so it must provide implementation of all seven methods from both **MouseListener** and **MouseMotionListener**. Notice, however, that there is no **addMouseInputListener** method, so we had to attach the same listener using separately **addMouseListener** and **addMouseMotionListener**.

Listing 110 MCN-Listeners/Listeners.java

```java
import java.awt.BorderLayout;
import java.awt.FlowLayout;
import java.awt.Font;
import java.awt.GridLayout;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.event.ActionListener;
import java.awt.event.MouseEvent;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.event.MouseInputListener;
import static javax.swing.SwingUtilities.*;

public class Listeners extends JFrame {
    public static void main (String[] args) {
        new Listeners();
    }

    private Listeners() {
        super("Listeners");
        setDefaultCloseOperation(EXIT_ON_CLOSE);
          // this is the default anyway...
        setLayout(new BorderLayout());

        JLabel lab = new JLabel(
            "Info on button clicks will appear here...",
            JLabel.CENTER);
        lab.setFont(new Font("SansSerif", Font.PLAIN, 20));
        JLabel mot = new JLabel(
            "Info on mouse motion will appear here...",
            JLabel.CENTER);
        mot.setFont(new Font("SansSerif", Font.PLAIN, 20));
        JPanel south = new JPanel(new GridLayout(2,1,10,10));
        south.add(lab);
        south.add(mot);
        add(south,BorderLayout.SOUTH);

        JPanel butts = new JPanel();
        butts.setLayout(new FlowLayout(FlowLayout.CENTER));
        ActionListener ac = e -> {
            lab.setText(
                    ((JButton)e.getSource()).getText() +
                                             " clicked");
        };
          // one listener for all buttons
        for (int i = 1; i < 4; ++i) {
            JButton b = new JButton("Button " + i);
            b.addActionListener(ac);
            butts.add(b);
        }
        add(butts,BorderLayout.NORTH);

        JTextArea area = new JTextArea(15,30);
        area.setFont(new Font("SansSerif",Font.PLAIN,18));
        area.setEditable(false);
        JScrollPane scroll = new JScrollPane(area,
                JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
                JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED);
        JPanel panel = new JPanel();
        panel.setPreferredSize(new Dimension(400,300));
        panel.setBackground(Color.BLUE);

        MouseInputListener inp = new MListener(area,mot);
        panel.addMouseListener(inp);
        panel.addMouseMotionListener(inp);
        JPanel panelCenter = new JPanel();
        panelCenter.setLayout(new GridLayout(2,1,5,10));
        panelCenter.add(panel);
        panelCenter.add(scroll);

        add(panelCenter,BorderLayout.CENTER);

        pack();
        setLocationRelativeTo(null);
        setVisible(true);
    }
}

class MListener implements MouseInputListener {
    private JTextArea area;
    private JLabel mot;
    public MListener(JTextArea a, JLabel m) {
        area = a;
        mot  = m;
    }
      // just a helper method
    private String info(MouseEvent e) {
        String which = " - UNKNOWN BUTTON!!!";
```

```
          // static methods imported from SwingUtilities
        if (isLeftMouseButton(e))   which = "left";
        if (isRightMouseButton(e))  which = "right";
        if (isMiddleMouseButton(e)) which = "middle";
        if (e.getClickCount() > 1)
            which += " (double click)";
        return " at x = " + e.getX() + " y = " + e.getY() +
                                      " (" + which + ")\n";
    }

    // MouseListener
    @Override
    public void mousePressed(MouseEvent e) {
        area.append("pressed" + info(e));
    }

    @Override
    public void mouseReleased(MouseEvent e) {
        area.append("released" + info(e));
    }

    @Override
    public void mouseClicked(MouseEvent e) {
        area.append("clicked" + info(e));
    }

    @Override
    public void mouseEntered(MouseEvent e) {
        area.append("entered\n");
    }

    @Override
    public void mouseExited(MouseEvent e) {
        area.append("exited\n");
    }

      // MouseMotionListener
    @Override
    public void mouseMoved(MouseEvent e) {
        mot.setText("  Moved: X=" + e.getX() +
                         ", Y=" + e.getY());
    }
    @Override
    public void mouseDragged(MouseEvent e) {
        mot.setText("Dragged: X=" + e.getX() +
                         ", Y=" + e.getY());
    }
}
```

10.4 Key events

The keyboard is a source of ‘physical’ events of type **KeyEvent**. These can be handled by listeners implementing **KeyListener**, which declares three methods:

```
public interface KeyListener extends EventListener {
    public void keyPressed(KeyEvent e);
    public void keyReleased(KeyEvent e);
    public void keyTyped(KeyEvent e);
}
```

These functions will be called

- `keyPressed` — when any key of the keyboard is pressed. The event passed as argument will contain information about which key it was, whether ***Shift***, ***Ctrl*** or ***Alt*** buttons were also pressed simultaneously, etc;
- `keyReleased` — when any key of the keyboard is released. The event passed as argument will, of course, contain information about the context of the event;
- `keyTyped` — when any key *corresponding to a Unicode character* is typed (pressed and released); this method will be invoked *after* **keyPressed** but before **keyReleased**. Pressing keys which *do not* correspond to Unicode characters will *not* trigger this method.

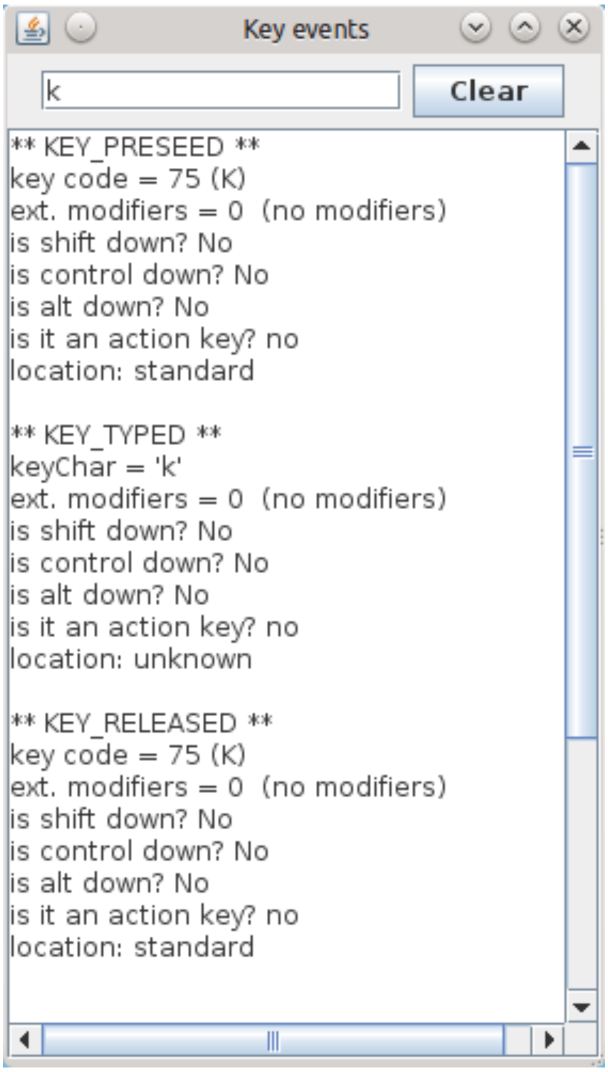
Let us consider the following example. It demonstrates what information one can extract from **KeyEvent**s passed to the listener’s methods:

Listing 111 MCT-KeyStrokes/KeyStrokes.java

```
import java.awt.BorderLayout;
import java.awt.FlowLayout;
import java.awt.Container;
import java.awt.Dimension;
import java.awt.event.ActionEvent;
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;
import javax.swing.AbstractAction;
import javax.swing.Action;
import javax.swing.JButton;
import javax.swing.JPanel;
import javax.swing.JTextArea;
import javax.swing.JTextField;
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.KeyStroke;

public class KeyStrokes extends JFrame
                        implements KeyListener {
    JTextField tField = null;
    JTextArea tArea = null;

    public static void main(String[] args) {
        new KeyStrokes();
```

```java
    }

    KeyStrokes() {
        super("Key events");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        tField = new JTextField(15);
        tField.addKeyListener(this);

        tArea = new JTextArea(40,25);
        tArea.setEditable(false);
        JScrollPane scroll = new JScrollPane(tArea);

        JButton clear = new JButton(
            new AbstractAction("Clear") {
                @Override
                public void actionPerformed(ActionEvent e) {
                    tArea.setText("");
                    tField.setText("");
                    tField.requestFocus();
                }
            }
        );

        JPanel north = new JPanel(new FlowLayout());
        north.add(tField);
        north.add(clear);

        add(north, BorderLayout.NORTH);
        add(scroll, BorderLayout.CENTER);

        pack();
        setLocationRelativeTo(null);
        setVisible(true);
    }

    public void keyTyped(KeyEvent e)    { info(e); }
    public void keyPressed(KeyEvent e)  { info(e); }
    public void keyReleased(KeyEvent e) { info(e); }

    private void info(KeyEvent e){
        int id = e.getID(); // type of the event
        String inf = null;
        if (id == KeyEvent.KEY_TYPED) {
            inf = "** KEY_TYPED **\n";
              // keyChar only defined for KEY TYPED,
              // i.e., for keys corresponding to
              // Unicode code points
            inf += "keyChar = '" +
                   e.getKeyChar() + "'\n";
        } else {
            if (id == KeyEvent.KEY_PRESSED)
                inf = "** KEY_PRESEED **\n";
            else
                inf = "** KEY_RELEASED **\n";
            int keyCode = e.getKeyCode();
            inf += "key code = " + keyCode + " (" +
                    KeyEvent.getKeyText(keyCode) + ")\n";
        }

        int modifs = e.getModifiersEx(); // extended...
        inf += "ext. modifiers = " + modifs + " ";
        String t = KeyEvent.getModifiersExText(modifs);
        if (t.length() > 0) inf += " (" + t + ")\n";
        else                inf += " (no modifiers)\n";

        inf += "is shift down? " +
                    (e.isShiftDown() ? "Yes\n" : "No\n");
        inf += "is control down? " +
                    (e.isControlDown() ?  "Yes\n" : "No\n");
        inf += "is alt down? " +
                    (e.isAltDown() ?  "Yes\n" : "No\n");

        inf += "is it an action key? " +
                (e.isActionKey() ? "yes\n" : "no\n");

        inf += "location: ";
        int loc = e.getKeyLocation();
        if (     loc == KeyEvent.KEY_LOCATION_STANDARD)
            inf += "standard\n\n";
        else if (loc == KeyEvent.KEY_LOCATION_LEFT)
            inf += "left\n\n";
        else if (loc == KeyEvent.KEY_LOCATION_RIGHT)
            inf += "right\n\n";
        else if (loc == KeyEvent.KEY_LOCATION_NUMPAD)
            inf += "numpad\n\n";
        else
            inf += "unknown\n\n";

        tArea.append(inf);
    }
}
```

which displays



```
** KEY_PRESEED **
key code = 75 (K)
ext. modifiers = 0  (no modifiers)
is shift down? No
is control down? No
is alt down? No
is it an action key? no
location: standard

** KEY_TYPED **
keyChar = 'k'
ext. modifiers = 0  (no modifiers)
is shift down? No
is control down? No
is alt down? No
is it an action key? no
location: unknown

** KEY_RELEASED **
key code = 75 (K)
ext. modifiers = 0  (no modifiers)
is shift down? No
is control down? No
is alt down? No
is it an action key? no
location: standard
```

The next two examples demonstrate handling both mouse and key events to 'free hand' drawing on a component:

Listing 112 MEA-DrawSimple/DrawIt.java

```java
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.awt.event.MouseMotionListener;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class DrawIt extends JFrame {

    public static void main(String[] args) {
        new DrawIt();
    }

    DrawIt() {
        setDefaultCloseOperation(EXIT_ON_CLOSE);

        MyPanel panel = new MyPanel();
          // panel will listen to mouse and keys
        panel.addMouseListener(panel);
        panel.addMouseMotionListener(panel);
        panel.addKeyListener(panel);

        panel.setBackground(Color.BLACK);
        //panel.setOpaque(true);
        panel.setPreferredSize(new Dimension(300,200));
        add(panel);

        pack();
        setVisible(true);
        setLocationRelativeTo(null);
    }
}

class MyPanel extends JPanel
        implements MouseListener,
                   MouseMotionListener,
                   KeyListener {
    int xOLD, yOLD;
    Graphics  pg;
    Color fore = Color.YELLOW;

      // MouseListener implementation
    public void mousePressed(MouseEvent e) {
        xOLD = e.getX();
        yOLD = e.getY();
        pg.setColor(fore);
    }
    public void mouseReleased(MouseEvent ignore) { }
    public void mouseClicked(MouseEvent  ignore) { }
    public void mouseEntered(MouseEvent  ignore) { }
    public void mouseExited(MouseEvent   ignore) { }

      // MouseMotionListener implementation
    public void mouseDragged(MouseEvent e)  {
        int x = e.getX(), y = e.getY();
        pg.setColor(fore);
        pg.drawLine(xOLD,yOLD,x,y);
        xOLD = x;
        yOLD = y;
    }
    public void mouseMoved(MouseEvent ignore)  { }

      // KeyListener implementation
```

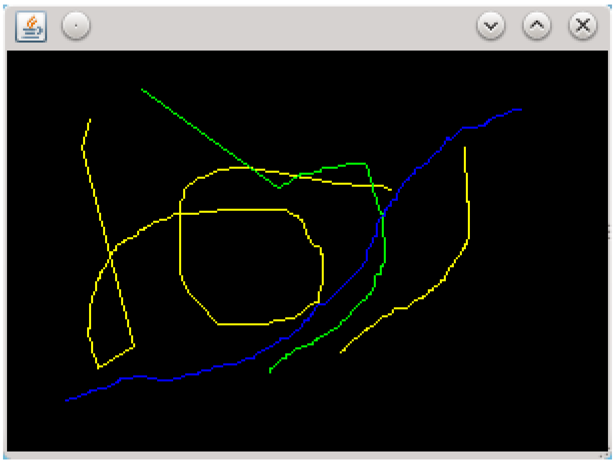
```java
    public void keyPressed(KeyEvent e)  {
        switch (e.getKeyCode()) {
            case KeyEvent.VK_R: fore = Color.RED;    break;
            case KeyEvent.VK_G: fore = Color.GREEN;  break;
            case KeyEvent.VK_B: fore = Color.BLUE;   break;
            case KeyEvent.VK_Y: fore = Color.YELLOW; break;
        }
    }
    public void keyReleased(KeyEvent ignore) { }
    public void keyTyped(KeyEvent    ignore) { }

    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        if (pg != null) pg.dispose();
        pg = getGraphics();
        requestFocus();     // so panel listens to keys...
    }
}
```

The approach presented above is not perfect, because when the component is redrawn, all its contents disappears. We can correct it by remembering all what we have drawn so far and recreating it every time the component is being redrawn:

Listing 113 MED-DrawBetter/DrawBetter.java

```java
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.awt.event.MouseMotionListener;
import java.util.ArrayList;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class DrawBetter extends JFrame {

    public static void main(String[] args) {
        new DrawBetter();
    }

    DrawBetter() {
        setDefaultCloseOperation(EXIT_ON_CLOSE);

        MyPanelBetter panel = new MyPanelBetter();
          // panel will listen to mouse and keys
        panel.addMouseListener(panel);
        panel.addMouseMotionListener(panel);
        panel.addKeyListener(panel);

        panel.setBackground(Color.BLACK);
        panel.setOpaque(true);
        panel.setPreferredSize(new Dimension(300,200));
        add(panel);

        pack();
        setLocationRelativeTo(null);
        setVisible(true);
    }
}

class MyPanelBetter extends JPanel
        implements MouseListener,
                   MouseMotionListener,
                   KeyListener {
    int xOLD, yOLD;
    Graphics  pg;
    Color fore = Color.YELLOW;
    ArrayList<Segment> segs = new ArrayList<Segment>();

      // MouseListener implementation
    public void mousePressed(MouseEvent e) {
        xOLD = e.getX();
        yOLD = e.getY();
        pg.setColor(fore);
    }
    public void mouseReleased(MouseEvent ignore) { }
    public void mouseClicked(MouseEvent  ignore) { }
    public void mouseEntered(MouseEvent  ignore) { }
    public void mouseExited(MouseEvent   ignore) { }

      // MouseMotionListener implementation
    public void mouseDragged(MouseEvent e)  {
        int x = e.getX(), y = e.getY();
        pg.drawLine(xOLD,yOLD,x,y);
        segs.add(new Segment(fore,x,y,xOLD,yOLD));
        xOLD = x;
        yOLD = y;
    }
    public void mouseMoved(MouseEvent ignore)  { }

      // KeyListener implementation
    public void keyPressed(KeyEvent e)  {
        switch (e.getKeyCode()) {
            case KeyEvent.VK_R: fore = Color.RED;    break;
            case KeyEvent.VK_G: fore = Color.GREEN;  break;
            case KeyEvent.VK_B: fore = Color.BLUE;   break;
            case KeyEvent.VK_Y: fore = Color.YELLOW; break;
        }
    }
    public void keyReleased(KeyEvent ignore) { }
    public void keyTyped(KeyEvent    ignore) { }

    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        if (pg != null) pg.dispose();
        pg = getGraphics();
        for (Segment s : segs)
            s.drawYourself(g);
        requestFocus();     // so panel listens to keys...
    }
}

class Segment {
    private Color color;
    private int x,y,xOLD,yOLD;

    Segment(Color c, int xn, int yn, int xo, int yo) {
        color = c;
        x     = xn;
        y     = yn;
        xOLD  = xo;
        yOLD  = yo;
    }
    void drawYourself(Graphics g) {
        g.setColor(color);
        g.drawLine(xOLD,yOLD,x,y);
    }
}
```

which displays



And one more example, again demonstrating key events:

Listing 114 MEG-MovFigure/MoveFig.java

```java
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.RenderingHints;
import java.awt.event.KeyAdapter;
import java.awt.event.KeyEvent;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class MoveFig extends JFrame {

    public static void main(String[] args) {
        new MoveFig();
    }

    public MoveFig() {
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setContentPane(new Figure());
        pack();
        setLocationRelativeTo(null);
        setVisible(true);
    }
}

class Figure extends JPanel {

    private final int step = 4, maxR = 150, minR = 10;
    private int radius = 10, xcoor = 10, ycoor = 10;
    private int width,height;
    private Color kolor = Color.red;
    private boolean blk = false, cir = true, fil = true;

    Figure() {
        setBackground(Color.black);
        setPreferredSize(new Dimension(400,400));

        addMouseListener(new MouseAdapter()
        {
            public void mousePressed(MouseEvent e)
            {
                if ( e.isMetaDown() ) {
                    blk = true;
                } else {
                    xcoor  = e.getX();
                    ycoor  = e.getY();
                    blk = false;
                }
                repaint();
            }
        });

        addKeyListener(new KeyAdapter()
        {
            public void keyPressed(KeyEvent e)
            {
                switch ( e.getKeyCode() ) {
                    case KeyEvent.VK_UP :
                        ycoor = (height+ycoor-step)%height;
                        break;
                    case KeyEvent.VK_DOWN :
                        ycoor = (ycoor+step)%height;
                        break;
                    case KeyEvent.VK_LEFT :
                        xcoor = (width+xcoor-step)%width;
                        break;
                    case KeyEvent.VK_RIGHT :
                        xcoor = (xcoor+step)%width;
                        break;
                    case KeyEvent.VK_ADD :
                        radius = Math.min(maxR,radius+step);
                        break;
                    case KeyEvent.VK_SUBTRACT :
                        radius = Math.max(minR,radius-step);
                        break;
                    case KeyEvent.VK_ENTER :
                        cir = !cir;
                        break;
                    case KeyEvent.VK_R :
                        kolor = Color.red;
                        break;
                    case KeyEvent.VK_G :
                        kolor = Color.green;
                        break;
                    case KeyEvent.VK_B :
                        kolor = Color.blue;
                        break;
                    case KeyEvent.VK_Y :
                        kolor = Color.yellow;
                        break;
                    case KeyEvent.VK_SPACE :
                        fil = !fil;
                        break;
                    default:
                        return;
                }
```

```java
                repaint();
            }
        });
    }

    public void paintComponent(Graphics g) {

        Graphics2D g2 = (Graphics2D)g;
        g2.setRenderingHint(
            RenderingHints.KEY_ANTIALIASING,
            RenderingHints.VALUE_ANTIALIAS_ON);
        g2.setRenderingHint(
            RenderingHints.KEY_TEXT_ANTIALIASING,
            RenderingHints.VALUE_TEXT_ANTIALIAS_ON);
        super.paintComponent(g);
        requestFocus();

        int x, y, b;

        width  = getWidth();
        height = getHeight();

        if ( !blk ) {
            g.setColor(kolor);
            x = xcoor - radius;
            y = ycoor - radius;
            b = 2*radius;
            if ( cir ) {
                if ( fil ) { g.fillOval(x,y,b,  b);   }
                else       { g.drawOval(x,y,b-1,b-1); }
            } else {
                if ( fil ) { g.fillRect(x,y,b,  b  ); }
                else       { g.drawRect(x,y,b-1,b-1); }
            }
        }
    }
}
```

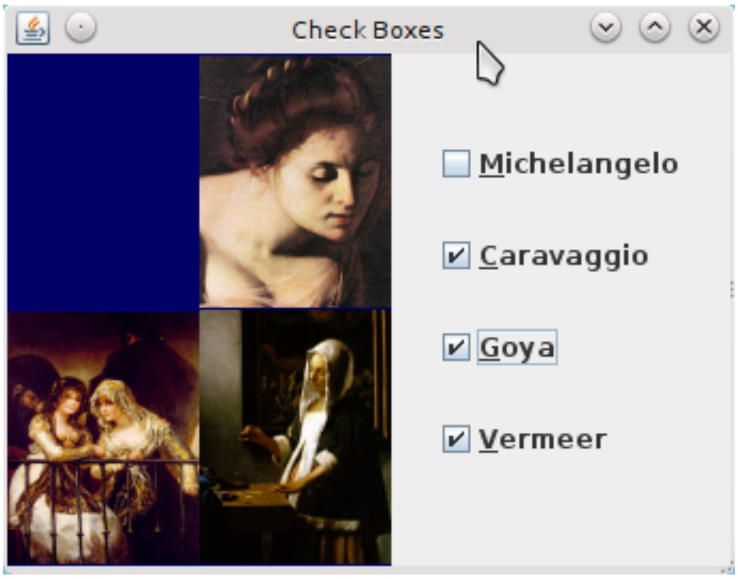
The last example shows another very useful component — a check box. Consider the following program:

Listing 115 MDI-CheckBoxes/CheckBoxes.java

```java
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.Image;
import java.awt.event.ItemEvent;
import java.awt.event.ItemListener;
import java.awt.event.KeyEvent;
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;
import javax.imageio.ImageIO;
import javax.swing.Box;
import javax.swing.BoxLayout;
import javax.swing.BorderFactory;
import javax.swing.JCheckBox;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class CheckBoxes {
    static String[] names = {"Michelangelo", "Caravaggio",
                             "Goya",         "Vermeer"};
    static BufferedImage[] images = new BufferedImage[4];
    static int[] mnemos = {KeyEvent.VK_M, KeyEvent.VK_C,
                           KeyEvent.VK_G, KeyEvent.VK_V};
    boolean[] states = {false,false,false,false};
    ImagePanel  imgPanel = new ImagePanel();

    public static void main(String[] args) {
        try {
            for (int i = 0; i < 4; ++i)
                images[i] = ImageIO.read(
                        new File(names[i]+".jpg"));
        } catch (IOException e) {
            System.out.println("Problems with images...");
            return;
        }
        new CheckBoxes();
    }

    private CheckBoxes() {
        JFrame f = new JFrame("Check Boxes");
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f.add(new ButtonPanel(), BorderLayout.EAST);
        f.add(imgPanel, BorderLayout.CENTER);
        f.pack();
        f.setLocationRelativeTo(null);
        f.setVisible(true);
    }

    class ButtonPanel extends JPanel {
        private final Dimension rig = new Dimension(0,20);
        ButtonPanel() {
            setLayout(new BoxLayout(this,BoxLayout.Y_AXIS));
            ItemListener lis = new MyItemListener();
            for (int i = 0; i < 4; ++i) {
```

```java
                add(Box.createRigidArea(rig));
                JCheckBox b = new JCheckBox(names[i]);
                b.setMnemonic(mnemos[i]);
                b.setSelected(false);
                b.addItemListener(lis);
                b.putClientProperty("i",Integer.valueOf(i));
                add(b);
            }
            setBorder(BorderFactory.createEmptyBorder(
                                          20,20,20,20));
        }
    }

    class ImagePanel extends JPanel {
        ImagePanel() {
            setPreferredSize(new Dimension(300,500));
            setBackground(new Color(0,0,102));
            setOpaque(true);
        }
        @Override
        public void paintComponent(Graphics g) {
            super.paintComponent(g);
            int panW = getWidth(),
                panH = getHeight();
            Image im = null;
            for (int i = 0; i < 4; ++i) {
                if (!states[i]) continue;
                Image img = images[i];
                int oriW = img.getWidth(null),
                    oriH = img.getHeight(null);
                if (panW*oriH < panH*oriW)
                    im = img.getScaledInstance(panW/2, -1,
                                       Image.SCALE_SMOOTH);
                else
                    im = img.getScaledInstance(-1, panH/2,
                                       Image.SCALE_SMOOTH);
                int imgW = im.getWidth(null);
                int imgH = im.getHeight(null);
                g.drawImage(im,(i%2)*panW/2+(panW/2-imgW)/2,
                               (i/2)*panH/2+(panH/2-imgH)/2,
                            imgW,imgH,null);
            }
        }
    }

    class MyItemListener implements ItemListener {
        @Override
        public void itemStateChanged(ItemEvent e) {
            JCheckBox source =
                        (JCheckBox)e.getItemSelectable();
            int i = (Integer)source.getClientProperty("i");
            if (e.getStateChange() == ItemEvent.DESELECTED)
                states[i] = false;
            else
                states[i] = true;
            imgPanel.repaint();
        }
    }
}
```

which displays



10.5 Window events

Closing, resizing, activating a window also generates events that can be handled. Listeners of these events have to implement the **WindowListener** interface with seven methods (as in the case of mouse listeners, there is the **WindowAdapter** class which provides empty implementation of all these methods):

```
public interface WindowListener extends EventListener {
    public void windowActivated(WindowEvent e);
    public void windowDeactivated(WindowEvent e);
    public void windowClosing(WindowEvent e);
    public void windowClosed(WindowEvent e);
    public void windowIconified(WindowEvent e);
    public void windowDeiconified(WindowEvent e);
    public void windowOpened(WindowEvent e);
}
```

These functions will be called

- `windowActivated` — when a window is set to be the active window;
- `windowDeactivated` — when a window becomes not the active window;
- `windowClosing` — when the tries to close the window, but before actually closing it;

- `windowClosed` — when the window has been closed;
- `windowIconified` — when the window changes its state from normal to minimized;
- `windowDeiconified` — when the changes its state from minimized to normal;
- `windowOpen` — when the window is made visible for the first time.

Among these methods, the third, **windowClosing**, is probably used most often — as in the example below:

Listing 116 MEO-WinClose/WinClose.java

```java
import java.awt.Dimension;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.JFrame;
import javax.swing.JOptionPane;

public class WinClose {
    public static void main(String... args) {
        JFrame f = new JFrame("WinClose");
        f.setDefaultCloseOperation(
                JFrame.DO_NOTHING_ON_CLOSE);
        f.addWindowListener(new WindowAdapter() {
            @Override
            public void windowClosing(WindowEvent w) {
                int d = JOptionPane.showConfirmDialog(
                    f,"Really close?", "Closing...",
                    JOptionPane.YES_NO_OPTION,
                    JOptionPane.QUESTION_MESSAGE);
                if (d==JOptionPane.YES_OPTION) f.dispose();
            }
        });

        f.setSize(new Dimension(300,200));
        f.setLocationRelativeTo(null);
        f.setVisible(true);
    }
}
```

The example below demonstrates the use of all window events

Listing 117 MER-WinEvents/WindowEvents.java

```java
import java.awt.Font;
import java.awt.event.WindowEvent;
import java.awt.event.WindowFocusListener;
import java.awt.event.WindowListener;
import java.awt.event.WindowStateListener;
import javax.swing.JFrame;
import javax.swing.JOptionPane;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import static javax.swing.JFrame.*;

public class WindowEvents extends JFrame
        implements WindowListener,
                   WindowFocusListener,
                   WindowStateListener {
    JTextArea tArea;

    public static void main(String[] args) {
        new WindowEvents();
    }

    WindowEvents() {
        super("WindowEvents");
        setDefaultCloseOperation(DO_NOTHING_ON_CLOSE);
        tArea = new JTextArea(40,42);
        tArea.setFont(new Font("Monospaced",Font.PLAIN,14));
        tArea.setEditable(false);
        JScrollPane scrollPane = new JScrollPane(tArea);
        add(scrollPane);

        addWindowListener(this);
        addWindowFocusListener(this);
        addWindowStateListener(this);

        pack();
        setLocationRelativeTo(null);
        setVisible(true);
    }

      // WindowListener;
    @Override
    public void windowActivated(WindowEvent e) {
        info("WindowListener::windowActivated");
    }
    @Override
    public void windowClosed(WindowEvent e) {
          // only seen on standard output
        info("WindowListener::windowClosed");
    }
    @Override
    public void windowClosing(WindowEvent e) {
        info("WindowListener::windowClosing");
        int answer = JOptionPane.showConfirmDialog(
            this,"Really quit?","Please, confirm",
            JOptionPane.YES_NO_OPTION,
            JOptionPane.QUESTION_MESSAGE
        );
        if (answer == JOptionPane.YES_OPTION) dispose();
    }
    @Override
    public void windowDeactivated(WindowEvent e) {
        info("WindowListener::windowDeactivated");
    }
    @Override
    public void windowDeiconified(WindowEvent e) {
        info("WindowListener::windowDeiconified");
    }
    @Override
    public void windowIconified(WindowEvent e) {
        info("WindowListener::windowIconified");
    }
    @Override
    public void windowOpened(WindowEvent e) {
        info("WindowListener::windowOpened");
    }

      // WindowFocusListener
    @Override
    public void windowGainedFocus(WindowEvent e) {
        info("WindowFocusListener::windowGainedFocus");
    }
    @Override
    public void windowLostFocus(WindowEvent e) {
        info("WindowFocusListener::windowLostFocus");
    }

      // WindowStateListener
    public void windowStateChanged(WindowEvent e) {
        infoState(e);
    }

    private void info(String msg) {
        tArea.append(msg + "\n");
        System.out.println(msg);
    }

    private void infoState(WindowEvent e) {
        int newState = e.getNewState();
        int oldState = e.getOldState();
        String msg = "WindowStateListener::"
                +  "windowStateChanged"
                + "\n***Changing state: old "
                + stateToString(oldState)
                + "\n                   new "
                + stateToString(newState);
        info(msg);
    }

    private String stateToString(int state) {
        if (state == JFrame.NORMAL) return "NORMAL";

        String strState = "";
        if ((state & JFrame.ICONIFIED) != 0) {
            strState += "ICONIFIED";
        }
        // constants from static import
        if ((state & MAXIMIZED_BOTH) == MAXIMIZED_BOTH) {
            strState += "MAXIMIZED_BOTH";
        } else {
            if ((state & MAXIMIZED_VERT) != 0) {
                strState += "MAXIMIZED_VERT";
            }
            if ((state & MAXIMIZED_HORIZ) != 0) {
                strState += "MAXIMIZED_HORIZ";
            }
        }
        if ("".equals(strState)) strState = "UNKNOWN";

        return strState;
    }
}
```

10.6 *Events and listeners — summary*

The table below presents type of events that can be handled by appropriate listeners. Each event has an identifier — an **int** that can be obtained by invoking **getID** method; it returns one of the constants which is given in the second column. The third column contains names of methods that have to be implemented by a listener of the type given in the fourth column:

Table 1: Events, listeners and their methods

Event	**ID** (int)	**Methods**	**Listener**
ActionEvent	ACTION_PERFORMED	actionPerformed	ActionListener
AdjustmentEvent	ADJUSTMENT_VALUE_CHANGED	adjustmentValueChanged	AdjustmentListener
ItemEvent	ITEM_STATE_CHANGED	itemStateChanged	ItemListener
TextEvent	TEXT_VALUE_CHANGED	textValueChanged	TextListener
MouseEvent	MOUSE_ENTERED MOUSE_EXITED MOUSE_PRESSED MOUSE_RELEASED MOUSE_CLICKED	mouseEntered mouseExited mousePressed mouseReleased mouseClicked	MouseListener or MouseInputListener
	MOUSE_MOVED MOUSE_DRAGGED	mouseMoved mouseDragged	MouseMotionListener or MouseInputListener
MouseWheelEvent	MOUSE_WHEEL_MOVED	mouseWheelMoved	MouseWheelListener

Table 1: Events, listeners and their methods (continued)

Event	**ID** (int)	**Methods**	**Listener**
KeyEvent	KEY_PRESEED KEY_RELEASED KEY_TYPED	keyPressed keyReleased keyTyped	KeyListener
FocusEvent	FOCUS_GAINED FOCUS_LOST	focusGained focusLost	FocusListener
ContainerEvent	COMPONENT_ADDED COMPONENT_REMOVED	componentAdded componentRemoved	ContainerListener
ComponentEvent	COMPONENT_HIDDEN COMPONENT_SHOWN COMPONENT_MOVED COMPONENT_RESIZED	componentHidden componentShown componentMoved componentResized	ComponentListener
WindowEvent	WINDOW_ACTIVATED WINDOW_DEACTIVATED WINDOW_ICONIFIED WINDOW_DEICONIFIED WINDOW_OPENED WINDOW_CLOSING WINDOW_CLOSED	windowActivated windowDeactivated windowIconified windowDeiconified windowOpened windowClosing windowClosed	WindowListener
	WINDOW_STATE_CHANGED	windowStateChanged	WindowStatelListener
	WINDOW_GAINED_FOCUS WINDOW_LOST_FOCUS	windowGainedFocus windowLostFocus	WindowFocusListener

The next table shows various types of events with their possible sources:

Table 2: Events and their sources

Event	**Source**
ComponentEvent	all components of AWT and Swing
ContainerEvent	AWT containers, Box container and all JComponents
MouseEvent	all components of AWT and Swing
FocusEvent	all components of AWT and Swing which can own the focus (for which this feature has not been disabled)
KeyEvent	all components of AWT and Swing which can own the focus
WindowEvent	all components derived from Window
InternalFrameEvent	internal windows of Swing
PropertyChangeEvent	all components of AWT and Swing
ActionEvent	components of type Button, JButton, JToggleButton, JCheckBox, JRadioButton, MenuItem, JMenuItem, CheckBoxMenuItem, JCheckBoxMenuItem, JRadioButtonMenuItem, TextField, JTextField, List, JComboBox
TextEvent	components of type TextField, TextArea (AWT only)

Table 2: Events and their sources (continued)

Event	**Source**
ItemEvent	components of type CheckBox, CheckBoxMEnuItem, JToggleButton, JCheckBox, JRadioButton, JCheckBoxMenuItem, JRadioButtonMenuItem, List, JComboBox
AdjustingEvent	components of type ScrollBar and JScrollBar

Section 11

GUI — models

11.1 MVC

The graphical system of Java, *swing* library in particular, uses the so called MVC (**M**odel–**V**iew–**C**ontroller) architecture, inspired by the language Smalltalk.
For a graphical component

- its *model* defines the data associated with the component: e.g., for a **JButton**, it can be information whether it is enabled or not, its action command etc., for a **JList** — its size and individual elements;
- its *view* specifies a visual representation of the component (size, color, borders etc.);
- its *controller* provides an interaction between the user and the view and between the view and the model.

Such an separation allows for more flexible code, because each part can be treated independently with well defined means of transferring information between them. Ideally, the model is completely independent of the view and, for example, one model may be used by several different views. In practice, however, Java (more specifically Swing) combines the view and the controller into one hybrid part, the so called UI (**U**ser **I**nterface) delegate.

Almost all Swing components are backed by their appropriate models — objects of classes implementing special interfaces. Generally, there are two types of models:

- GUI models which correspond to visual features of components (a button pressed or not, an item selected or not);
- data models which correspond to pure data without referring to any visual features.

The table below shows models used by various components of Swing:

Table 3: Components and their models

Component	Model interface	Type
JButton	ButtonModel	GUI
JToggleButton	ButtonModel	GUI/data
JCheckBox	ButtonModel	GUI/data
JRadioButton	ButtonModel	GUI/data
JMenu	ButtonModel	GUI
JMenuItem	ButtonModel	GUI
JCheckBoxMenuItem	ButtonModel	GUI/data
JRadioButtonMenuItem	ButtonModel	GUI/data
JComboBox	ComboBoxModel	data
JProgressBar	BoundedRangeModel	GUI/data
JScrollBar	BoundedRangeModel	GUI/data

Table 3: Components and their models (cont.)

Component	Model interface	Type
JSlider	BoundedRangeModel	GUI/data
JTabbedPane	SingleSelectionModel	GUI
JList	ListModel	data
JList	ListSelectionModel	GUI
JTable	TableModel	data
JTable	TableColumnModel	GUI
JTree	TreeModel	data
JTree	TreeSelectionModel	GUI
JEditorPane	Document	data
JTextPane	Document	data
JTextArea	Document	data
JTextField	Document	data
JPasswordField	Document	data

Many Swing components create their models automatically; for example, we have been using JButtons without even knowing that they also have models. Sometimes those default models (their names usually start with Default, e.g., DefaultListModel is the default model implementing the ListModel interface) are sufficient and we don't have to deal with them explicitly in our programs. Still, we always have access to the underlying models by means of getModel/setModel methods of Swing components. This makes it possible to modify (configure) these models or replace them with our own, 'home made' model (object of a class implementing an appropriate interface). Sometimes components themselves expose some methods which in fact operate on the the underlying model, so we can use them without explicitly referring to models. For example, the class JSlider exposes methods which allow us to get or set the current position of the slider: in fact these method refer to an underlying model (of type BoundedRangeModel)

```
    slider.getValue()
```

is implemented as

```
    slider.getModel().getValue()
```

However, when dealing with more complex components, referring to models is inevitable.

11.2 Lists

The JList represents a widget displaying a list of objects. The underlying models that are used by the class are ListModel (pure data) and ListSelectionModel (related to GUI). In order to create simple lists and just display them, we don't have to refer to models, because, if not told otherwise, constructors will also create underlying models. One can use constructors which take an array of Objects or a Vector and the models will be created automatically:

```
    JList(Object[] arr)
```

or

```
JList(Vector vec)
```

Therefore, the following snippet will allow us to create a **JList** widget:

```
String[] arr = {"Cat", "Dog", "Cow"};
JList list = new JList(arr);
JScrollPane scroll = new JScrollPane(list);
```

which may then be displayed. However, the functionality of the resulting list will be somewhat limited: we cannot add or remove elements. Still, we *can* use several methods of **JList** which in fact delegate invocations to corresponding methods of the underlying models. For example,

```
list.getMinSelectedIndex()
```

is equivalent to

```
list.getSelectionModel().getMinSelectionIndex()
```

Also, instead of attaching listeners to models, we can attach them to a **JList** itself — a list selection listener attached to a list will in fact listen to changes of the model:

```
list.addListSelectionListener(lis);
```

is equivalent to

```
list.getSelectionModel().addListSelectionListener(lis);
```

(although in the first case **getSource** inside listeners' methods returns the list itself, while in the second case it will return the model).

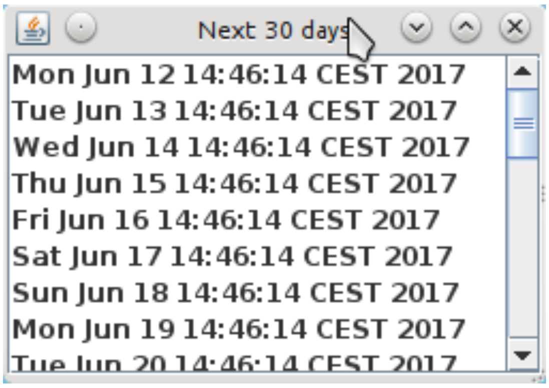
In order to be able to create more flexible lists (with dynamic contents), we need to implement a model (interface is **ListModel**) and then pass it to a constructor of **JList** (or invoke **setModel** on an existing list).

A list model must implement **ListModel**, but the most convenient way to create a custom model is to inherit from **AbstractListModel**. Then, there are only two simple methods to implement: **getSize** and **getElementAt(int)**. The implementation does not even need to hold the data in any kind of a collection, as in the example below:

Listing 118 PEJ-ListAdHoc/ListAdHoc.java

```java
import java.awt.Dimension;
import java.util.Calendar;
import java.util.Date;
import javax.swing.AbstractListModel;
import javax.swing.JFrame;
import javax.swing.JList;
import javax.swing.JScrollPane;

public class ListAdHoc extends JFrame {
    public static void main(String[] args) {
        new ListAdHoc();
    }
    ListAdHoc() {
        super("Next 30 days");
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        JScrollPane scroll = new JScrollPane(
                        new JList<Date>(new MyModel()));
        scroll.setPreferredSize(new Dimension(250,150));
        add(scroll);
        pack();
        setLocationRelativeTo(null);
        setVisible(true);
    }
}

class MyModel extends AbstractListModel<Date> {
    @Override
    public int getSize() { return 30; }
    public Date getElementAt(int i) {
        Calendar cal = Calendar.getInstance();
        cal.add(Calendar.DATE, i+1);
        return cal.getTime();
    }
}
```

which displays a widget like this



```
Mon Jun 12 14:46:14 CEST 2017
Tue Jun 13 14:46:14 CEST 2017
Wed Jun 14 14:46:14 CEST 2017
Thu Jun 15 14:46:14 CEST 2017
Fri Jun 16 14:46:14 CEST 2017
Sat Jun 17 14:46:14 CEST 2017
Sun Jun 18 14:46:14 CEST 2017
Mon Jun 19 14:46:14 CEST 2017
Tue Jun 20 14:46:14 CEST 2017
```

Modifications of a list (strictly speaking, its model) generate events of type ListDataEvent and convey information on

- the type of modification: interval (maybe consisting of only one item) of elements has been added, removed, changed;
- the lower index of the interval involved (getIndex0);
- the upper index of the interval involved (getIndex1);

These events will be passed to listener's methods declared in ListDataListener with three methods:

- `public void intervalAdded(ListDataEvent e)`
- `public void intervalRemoved(ListDataEvent e)`
- `public void contentsChanged(ListDataEvent e)`

When writing an implementation of ListModel, it is convenient to either use the class DefaultListModel (where all basic functionality is already implemented) or inherit from AbstractListModel and provide implementation for only two remaining abstract methods: getElementAt and getSize. If our list is to be modifiable, we add methods for adding and removing elements of the list, but then we have to remember to fire appropriate events which will inform listeners about the modifications. This is easy to do, because in AbstractListModel there are three methods (already implemented) that we can use:

- protected void fireIntervalAdded(Object source, int i0, int i1)
- protected void fireIntervalRemoved(Object source, int i0, int i1)
- protected void fireContentsChanged(Object source, int i0, int i1)

They generate events and call appropriate methods on all registered listeners.

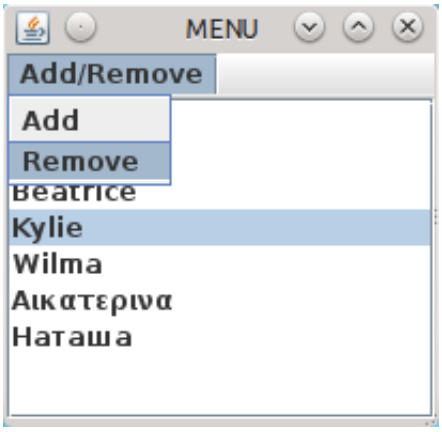
Let us consider a simple example. Here, we define our own implementation of ListModel; as the list is modifiable, we have to notify listeners about our modifications:

Listing 119 PEQ-ListMenu/ListWithMenu.java

```java
import java.awt.BorderLayout;
import java.awt.Component;
import java.awt.Dimension;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.AbstractListModel;
import javax.swing.JFrame;
import javax.swing.JList;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import java.util.Collections;
import java.util.ArrayList;

public class ListWithMenu {
    public static void main (String[] args) {
        JFrame f = new JFrame("LIST");
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JPanel panel = new JPanel();
        panel.setLayout(new BorderLayout());

        MyModel model = new MyModel(panel);
        JList<String> list = new JList<>(model);
        JScrollPane scroll = new JScrollPane(list);
        scroll.setPreferredSize(new Dimension(200,150));

        JMenu addRem = new JMenu("Add/Remove");
        JMenuItem add = new JMenuItem("Add");
        add.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                String s = JOptionPane.showInputDialog(
                    panel, "Enter name to add", "Add",
                    JOptionPane.QUESTION_MESSAGE);
                model.add(s);
            }
        });
        JMenuItem rem = new JMenuItem("Remove");
        rem.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                String s = JOptionPane.showInputDialog(
                    panel, "Enter name to remove", "Remove",
                    JOptionPane.QUESTION_MESSAGE);
                model.remove(s);
            }
        });
        addRem.add(add);
        addRem.add(rem);
        JMenuBar menuBar = new JMenuBar();
        menuBar.add(addRem);
        f.setJMenuBar(menuBar);

        f.add(scroll); // CENTER by default
        f.pack();
        f.setLocationRelativeTo(null);
        f.setVisible(true);
    }
}

class MyModel extends AbstractListModel<String> {
    ArrayList<String> data  = new ArrayList<String>();
    Component comp;

    MyModel(Component c) {
        comp = c;
        data.add("Alice");
        data.add("Kylie");
        data.add("Wilma");
    }

    public void add(String s) {
        if (data.contains(s)) {
            JOptionPane.showMessageDialog(
                comp, s + " exists", "ERROR",
                JOptionPane.ERROR_MESSAGE);
            return;
        }
        data.add(s);
        Collections.sort(data);
        fireIntervalAdded(this,0,data.size());
    }

    public void remove(String s) {
        if (!data.contains(s)) {
            JOptionPane.showMessageDialog(
                comp, "No " + s + " in list", "ERROR",
                JOptionPane.ERROR_MESSAGE);
            return;
```

```java
        }
        data.remove(s);
        fireIntervalRemoved(this,0,data.size());
    }

    @Override
    public String getElementAt(int i) {
        return data.get(i);
    }

    @Override
    public int getSize() {
        return data.size();
    }
}
```

the program displays a widget like this



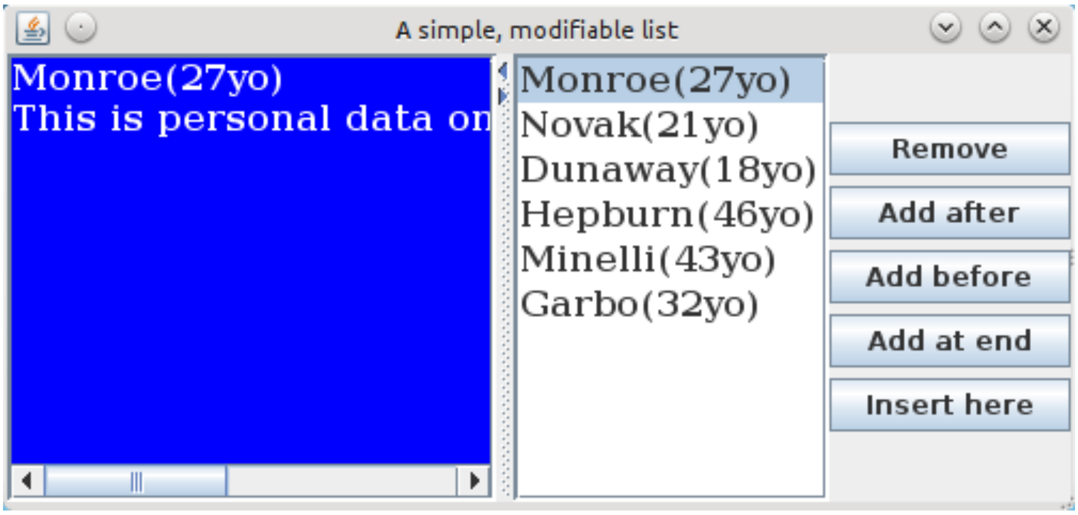
As we have said, lists have another model, **ListSelectionModel**, which holds information on whether an item of the list is, or is not, selected. An example below shows how we can use it (it also demonstrates the use of **JSplitPane** widget). In this program, we don't create our own implementation of the model — instead, we use the default implementation provided by **DefaultListModel**:

Listing 120 PEK-MVCList/SimpList.java

```java
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Component;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.event.ActionEvent;
import javax.swing.AbstractAction;
import javax.swing.Box;
import javax.swing.BoxLayout;
import javax.swing.DefaultListModel;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JList;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JSplitPane;
import javax.swing.JTextArea;
import javax.swing.ListSelectionModel;
import javax.swing.event.ListSelectionEvent;
import javax.swing.event.ListSelectionListener;

public class SimpList extends JFrame {
    public static void main(String[] args) {
        new SimpList();
    }

    SimpList() {
        super("A simple, modifiable list");
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        add(new MainPanel());
        pack();
        setLocationRelativeTo(null);
        setVisible(true);
    }
}

class MainPanel extends    JPanel
                implements ListSelectionListener {

    JTextArea                infoArea;
    DefaultListModel<Person> model;
    JList<Person>            list;
    int                      counter;

    MainPanel() {
        model = new DefaultListModel<>();
        model.addElement(new Person("Monroe",27));
        model.addElement(new Person("Novak",21));
        model.addElement(new Person("Dunaway",18));
        model.addElement(new Person("Hepburn",46));
        model.addElement(new Person("Minelli",43));
        model.addElement(new Person("Garbo",32));

        list = new JList<>(model);
        list.setSelectionMode(
                ListSelectionModel.SINGLE_SELECTION);
        list.setSelectedIndex(0);
        list.addListSelectionListener(this);
        list.setFont(new Font("Serif",Font.BOLD,16));

        JScrollPane scrollPers =
            new JScrollPane(list,
                JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED,
                JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED);

        infoArea = new JTextArea(10,14);
        infoArea.setFont(new Font("Serif",Font.BOLD,16));
        infoArea.setBackground(Color.blue);
        infoArea.setForeground(Color.white);
        JScrollPane scrollInfo =
            new JScrollPane(infoArea,
                JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED,
                JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED);

        JSplitPane split =
            new JSplitPane(JSplitPane.HORIZONTAL_SPLIT,
                           scrollInfo,scrollPers);
        split.setOneTouchExpandable(true);
        split.setDividerLocation(0.40);

        int   width = 200, height = 100;
        float xjust = CENTER_ALIGNMENT;

        JButton bRem = new JButton(
            new AbstractAction("Remove") {
                public void actionPerformed(ActionEvent e) {
                    if (list.isSelectionEmpty()) return;
                    int i = list.getSelectedIndex();
                    model.remove(i);
                    if (!model.isEmpty()) select(
                            Math.min(i,model.getSize()-1));
                }
            }
        );
        configButton(bRem,width,height,xjust);

        JButton bAddAfter = new JButton(
            new AbstractAction("Add after") {
                public void actionPerformed(ActionEvent e) {
                    if (list.isSelectionEmpty()) return;
                    int i = list.getSelectedIndex();
                    model.add(i+1,new Person("NewPerson" +
                                    ++counter,3*counter));
                    select(i);
                }
            }
        );
        configButton(bAddAfter,width,height,xjust);

        JButton bAddBefore =
            new JButton(new AbstractAction("Add before") {
                public void actionPerformed(ActionEvent e) {
```

```java
                    if (list.isSelectionEmpty()) return;
                    int i = list.getSelectedIndex();
                    model.add(i,new Person("NewPerson" +
                                    ++counter,3*counter));
                    select(i);
                }
            }
        );
        configButton(bAddBefore,width,height,xjust);

        JButton bAddEnd = new JButton(
            new AbstractAction("Add at end") {
                public void actionPerformed(ActionEvent e) {
                    model.addElement(new Person("NewPerson"+
                                    ++counter,3*counter));
                    select(model.getSize()-1);
                }
            }
        );
        configButton(bAddEnd,width,height,xjust);

        JButton bInsert = new JButton(
            new AbstractAction("Insert here") {
                public void actionPerformed(ActionEvent e) {
                    if (list.isSelectionEmpty()) return;
                    int i = list.getSelectedIndex();
                    model.set(i,new Person("NewPerson" +
                                    ++counter,3*counter));
                    select(i);
                }
            }
        );
        configButton(bInsert,width,height,xjust);

        JPanel box = new JPanel();
        Dimension rig = new Dimension(1,5);
        box.setLayout(new BoxLayout(box,BoxLayout.Y_AXIS));
        box.add(Box.createVerticalGlue());
        box.add(bRem);
        box.add(Box.createRigidArea(rig));
        box.add(bAddAfter);
        box.add(Box.createRigidArea(rig));
        box.add(bAddBefore);
        box.add(Box.createRigidArea(rig));
        box.add(bAddEnd);
        box.add(Box.createRigidArea(rig));
        box.add(bInsert);
        box.add(Box.createVerticalGlue());

        setLayout(new BorderLayout());
        add(box,  BorderLayout.EAST);
        add(split,BorderLayout.CENTER);

        select(0);
    }

    void configButton(JButton b, int w, int h, float just) {
        b.setMaximumSize(new Dimension(w,h));
        b.setAlignmentX(just);
    }

    void select(int i) {
        list.ensureIndexIsVisible(i);
        list.setSelectedIndex(i);
        infoArea.setText(
                ((Person)model.getElementAt(i)).getData());
        infoArea.setCaretPosition(0);
    }

    public void valueChanged(ListSelectionEvent e) {
        if (e.getValueIsAdjusting() ||
            list.isSelectionEmpty()) return;
        select(list.getSelectedIndex());
    }
}

class Person {
    private String name;
    private int    age;
    private String personalData;

    Person(String n, int a) {
        name = n;
        age  = a;
        personalData = name + "(" + age + "yo)\nThis is " +
            "personal data on the aforementioned person...";
    }
    String getData() {
        return personalData;
    }
    @Override
    public String toString() {
        return name + "(" + age + "yo)";
    }
}
```

The program displays:



Monroe(27yo)
This is personal data on

Monroe(27yo)
Novak(21yo)
Dunaway(18yo)
Hepburn(46yo)
Minelli(43yo)
Garbo(32yo)

Almost the same program may be rewritten with a custom implementation of the model (by extending AbstractListModel)

Listing 121 PEM-MVCListModif/CustomLModel.java

```java
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Component;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import javax.swing.AbstractAction;
import javax.swing.AbstractListModel;
import javax.swing.Box;
import javax.swing.BoxLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JList;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JSplitPane;
import javax.swing.JTextArea;
import javax.swing.ListSelectionModel;
import javax.swing.event.ListSelectionEvent;
import javax.swing.event.ListSelectionListener;
import static javax.swing.JScrollPane.*;

public class CustomLModel extends JFrame {
    public static void main(String[] args) {
        new CustomLModel();
    }

    CustomLModel() {
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setTitle("Example of modifiable list");
        Person[] data = {
                            new Person("Poe",27),
                            new Person("Elliot",21),
                            new Person("Whitman",18),
                            new Person("Pound",61),
                            new Person("Burns",43),
                            new Person("Byron",32),
                        };
        add(new MainPanel(data));

        pack();
        setLocationRelativeTo(null);
        setVisible(true);
    }
}

class MainPanel extends    JPanel
                implements ListSelectionListener {
    JTextArea     infoArea;
    LModel        model;
    JList<Person> list;
    int           count;

    MainPanel(Person[] arr) {
        model = new LModel(arr);
        list = new JList<Person>(model);
        list.setSelectionMode(
                ListSelectionModel.SINGLE_SELECTION);
        list.setSelectedIndex(0);
        list.addListSelectionListener(this);
        list.setFont(new Font("Serif",Font.BOLD,16));
        list.setPreferredSize(new Dimension(200,250));

        JScrollPane scrollPers =
            new JScrollPane(list,
                VERTICAL_SCROLLBAR_AS_NEEDED,
                HORIZONTAL_SCROLLBAR_AS_NEEDED);

        infoArea = new JTextArea(10,14);
        infoArea.setFont(new Font("Serif",Font.BOLD,16));
        infoArea.setBackground(Color.blue);
        infoArea.setForeground(Color.white);
        JScrollPane scrollInfo =
            new JScrollPane(infoArea,
                VERTICAL_SCROLLBAR_AS_NEEDED,
                HORIZONTAL_SCROLLBAR_AS_NEEDED);

        JSplitPane split =
            new JSplitPane(JSplitPane.HORIZONTAL_SPLIT,
                           scrollInfo,scrollPers);
        split.setOneTouchExpandable(true);
        split.setDividerLocation(0.40);

        int   width = 200, height = 100;
        float xjust = Component.CENTER_ALIGNMENT;

        JButton bDelete = new JButton(
            new AbstractAction("Delete") {
                public void actionPerformed(ActionEvent e) {
                    if (list.isSelectionEmpty()) return;
                    int i = list.getSelectedIndex();
                    model.remove(i);
                    if (!model.isEmpty()) select(
                            Math.min(i,model.getSize()-1));
                }
            }
        );
        configButton(bDelete,width,height,xjust);

        JButton bAddAfter = new JButton(
            new AbstractAction("Add after") {
                public void actionPerformed(ActionEvent e) {
                    if (list.isSelectionEmpty()) return;
                    int i = list.getSelectedIndex();
                    model.add(i+1,new Person("New person " +
                                    ++count,3*count));
                    select(i);
                }
            }
        );
        configButton(bAddAfter,width,height,xjust);

        JButton bAddBefore =
            new JButton(new AbstractAction("Add before") {
                public void actionPerformed(ActionEvent e) {
                    if (list.isSelectionEmpty()) return;
                    int i = list.getSelectedIndex();
                    model.add(i,new Person("New person " +
                                    ++count,3*count));
                    select(i);
                }
            }
        );
        configButton(bAddBefore,width,height,xjust);

        JButton bAddAtEnd = new JButton(
            new AbstractAction("Add at end") {
                public void actionPerformed(ActionEvent e) {
```

```java
                    model.add(new Person("New person " +
                                    ++count,3*count));
                    select(model.getSize()-1);
                }
            }
        );
        configButton(bAddAtEnd,width,height,xjust);

        JButton bInsert = new JButton(
            new AbstractAction("Replace") {
                public void actionPerformed(ActionEvent e) {
                    if (list.isSelectionEmpty()) return;
                    int i = list.getSelectedIndex();
                    model.set(i,new Person("New person " +
                                    ++count,3*count));
                    select(i);
                }
            }
        );
        configButton(bInsert,width,height,xjust);

        JPanel box = new JPanel();
        Dimension rig = new Dimension(1,5);
        box.setLayout(new BoxLayout(box,BoxLayout.Y_AXIS));
        box.add(Box.createVerticalGlue());
        box.add(bDelete);
        box.add(Box.createRigidArea(rig));
        box.add(bAddAfter);
        box.add(Box.createRigidArea(rig));
        box.add(bAddBefore);
        box.add(Box.createRigidArea(rig));
        box.add(bAddAtEnd);
        box.add(Box.createRigidArea(rig));
        box.add(bInsert);
        box.add(Box.createVerticalGlue());

        setLayout(new BorderLayout());
        add(box,  BorderLayout.EAST);
        add(split,BorderLayout.CENTER);

        select(0);
    }

    void configButton(JButton b, int w, int h, float just) {
        b.setMaximumSize(new Dimension(w,h));
        b.setAlignmentX(just);
    }

    void select(int i) {
        list.ensureIndexIsVisible(i);
        list.setSelectedIndex(i);
        infoArea.setText(
                ((Person)model.getElementAt(i)).getData());
        infoArea.setCaretPosition(0);
    }

      // implementing ListSelectionListener
    public void valueChanged(ListSelectionEvent e) {
        if (e.getValueIsAdjusting() ||
            list.isSelectionEmpty()) return;
        select(list.getSelectedIndex());
    }
}

class LModel extends AbstractListModel<Person> {

    List<Person> data;
    int size;

    LModel(Person[] arr) {
        data = new ArrayList<Person>(Arrays.asList(arr));
        size = data.size();
    }

    public void add(Person o) {
        data.add(o);
        fireIntervalAdded(this,size-1,size-1);
    }

    public void add(int i, Person p) {
        if (size > 0)
            data.add(i,p);
        else
            data.add(p);
        fireIntervalAdded(this,i,i);
    }

    public void remove(int i) {
        data.remove(i);
        fireIntervalRemoved(this,i,i);
    }

    public void set(int i, Person o) {
        data.set(i,o);
        fireContentsChanged(this,i,i);
    }

    public boolean isEmpty() {
        return data.isEmpty();
    }
```

```

      // interface ListModel
    @Override
    public Person getElementAt(int i) {
        return data.get(i);
    }
    @Override
    public int getSize() {
        return data.size();
    }
}

class Person {
    private String name;
    private int    age;
    private String data;

    Person(String n, int w) {
        name = n;
        age  = w;
        data = name + " " + age + "\nThis is\na descript" +
                    "ion of\nthis person:\nAddress: ...\n"+
                    "Telephone number: ...\ne\nt\nc...";
    }
    String getData() {
        return data;
    }
    @Override
    public String toString() {
        return name;
    }
}
```

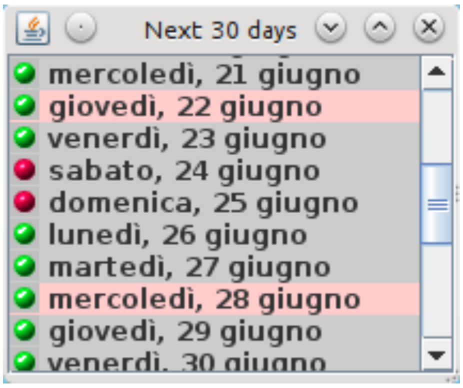
Individual cells are rendered as **JComponents** (if they are **JComponents**) or as **JLabels** with an appropriate strings as the text. We can, however, provide our own renderer: all we have to to is to implement the **ListCellRenderer** interface with one method. The program below demonstrates how to do it:

Listing 122 PEG-ListRender/ListRender.java

```
import java.awt.Dimension;
import java.awt.Color;
import java.awt.Component;
import java.util.Calendar;
import java.util.Locale;
import javax.swing.AbstractListModel;
import javax.swing.Icon;
import javax.swing.ImageIcon;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JList;
import javax.swing.JScrollPane;
import javax.swing.ListCellRenderer;
import static java.util.Calendar.*;

public class ListRender extends JFrame {
    public static void main(String[] args) {
        new ListRender();
    }
    ListRender() {
        super("Next 30 days");
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        JList<Calendar> list =  new JList<>(new MyModel());
        list.setCellRenderer(new CalRenderer());
        JScrollPane scroll = new JScrollPane(list);
        scroll.setPreferredSize(new Dimension(210,150));
        add(scroll);
        pack();
        setLocationRelativeTo(null);
        setVisible(true);
    }
}

class MyModel extends AbstractListModel<Calendar> {
    @Override
    public int getSize() { return 30; }
    public Calendar getElementAt(int i) {
        Calendar cal = Calendar.getInstance(Locale.ITALY);
        cal.add(Calendar.DATE, i+1);
        return cal;
    }
}

class CalRenderer extends JLabel
                  implements ListCellRenderer<Calendar> {
    static Color colUnSel = new Color(0xCC,0xCC,0xCC);
    static Color colSel   = new Color(0xFF,0xCC,0xCC);
    static Icon week = new ImageIcon("green.gif");
    static Icon holy = new ImageIcon("red.gif");
    static Locale loc = Locale.ITALY;

    @Override
    public Component getListCellRendererComponent (
            JList<? extends Calendar> list, Calendar cal,
            int index, boolean isSel, boolean hasFocus) {
        setOpaque(true);
        if (cal.get(DAY_OF_WEEK) == SUNDAY ||
            cal.get(DAY_OF_WEEK) == SATURDAY)
            setIcon(holy);
```

```java
        else
            setIcon(week);
        setBackground(isSel ? colSel : colUnSel);
        setText(cal.getDisplayName(DAY_OF_WEEK, LONG, loc) +
                ", " + cal.get(DAY_OF_MONTH) + " "+
                cal.getDisplayName(MONTH, LONG, loc));
        return this;
    }
}
```

The program dispalys:



mercoledì, 21 giugno
giovedì, 22 giugno
venerdì, 23 giugno
sabato, 24 giugno
domenica, 25 giugno
lunedì, 26 giugno
martedì, 27 giugno
mercoledì, 28 giugno
giovedì, 29 giugno
venerdì, 30 giugno

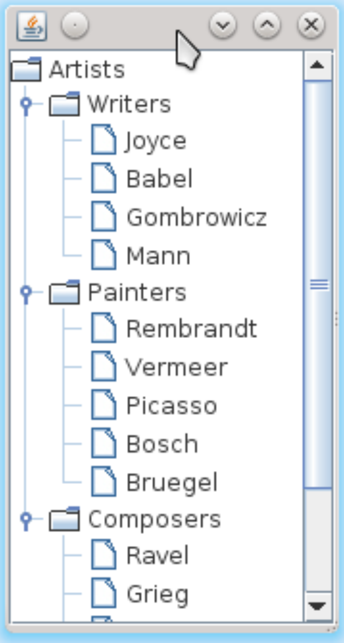
11.3 Trees

Another component that is often useful is **JTree**. It is backed by **TreeSelectionModel**. Listeners of this model implement the functional interface **TreeSelectionListener** with only one method — **valueChanged**. The example below demonstrates how to use basic features of trees:

Listing 123 PEI-JTree/MyTree.java

```java
import java.awt.Dimension;
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTree;
import javax.swing.event.TreeSelectionEvent;
import javax.swing.event.TreeSelectionListener;
import javax.swing.tree.DefaultMutableTreeNode;
import javax.swing.tree.TreeSelectionModel;

public class MyTree extends JFrame
                    implements TreeSelectionListener {
    public static void main(String[] args) {
        new MyTree();
    }

    String artists[] = {"Writers","Painters","Composers"};
    String data[][] =  {
        {"Joyce","Babel","Gombrowicz", "Mann" },
        {"Rembrandt","Vermeer","Picasso","Bosch","Bruegel"},
        {"Ravel","Grieg","Bach","Mahler","Mozart","Chopin"}
    };
    public MyTree() {
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        DefaultMutableTreeNode root =
                    new DefaultMutableTreeNode("Artists");
        for (int i = 0; i < artists.length; ++i) {
            DefaultMutableTreeNode n =
                    new DefaultMutableTreeNode(artists[i]);
            for (int j = 0; j < data[i].length; ++j)
                n.add(new DefaultMutableTreeNode(
                                     data[i][j]));
            root.add(n);
        }

        JTree tree = new JTree(root);
        tree.getSelectionModel().setSelectionMode
                (TreeSelectionModel.SINGLE_TREE_SELECTION);
        tree.addTreeSelectionListener(this);
        JScrollPane scroll = new JScrollPane(tree);
        scroll.setPreferredSize(new Dimension(170,300));
        add(scroll);
        pack();
        setLocationRelativeTo(null);
        setVisible(true);
    }
    @Override
    public void valueChanged(TreeSelectionEvent e) {
        JTree tree = (JTree) e.getSource();
        DefaultMutableTreeNode node =
                (DefaultMutableTreeNode)
                tree.getLastSelectedPathComponent();
        if (node == null) return;
        if (node.isLeaf())
            System.out.println(node.getUserObject());
    }
}
```

The program displays



Artists
Writers
Joyce
Babel
Gombrowicz
Mann
Painters
Rembrandt
Vermeer
Picasso
Bosch
Bruegel
Composers
Ravel
Grieg

However, trees are quite rich and complicated objects — further details can be found in the documentation. In particular, one can also define (as for lists and tables) custom cell renderers.

11.4 Models and listeners

As there are very many graphical widgets defined in Swing, the number of different models is also quite big: the table below shows commonly used models for various components:

Table 4: Model and their listeners

Model	**Listener interface**	**Type of events**
BoundedRangeModel	ChangeListener	ChangeEvent
ButtonModel	ChangeListener	ChangeEvent
SingleSelectionModel	ChangeListener	ChangeEvent
ListModel	ListDataListener	ListDataEvent
ListSelectionModel	ListSelectionListener	ListSelectionEvent
ComboBoxModel	ListDataListener	ListDataEvent
TreeModel	TreeModelListener	TreeModelEvent
TreeSelectionModel	TreeSelectionListener	TreeSelectionEvent
TableModel	TableModelListener	TableModelEvent
TableColumnModel	TableColumnModelListener	TableColumnModelEvent
Document	DocumentListener	DocumentEvent
Document	UndoableEditListener	UndoableEditEvent

11.5 Examples

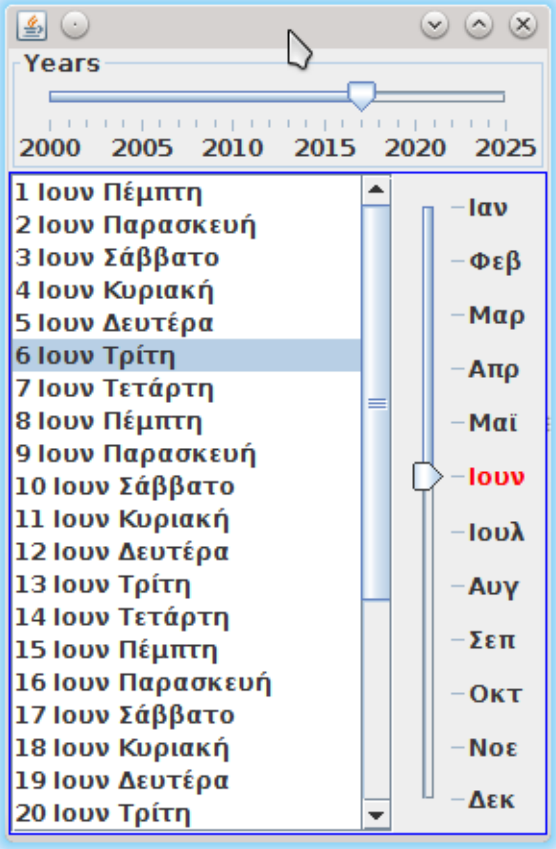
Let us now consider another, more complex, example of lists. Here, we also demonstrate, among other things, **JSlider** widget:

Listing 124 MEM-JListSlider/SliList.java

```java
import java.awt.BorderLayout;
import java.awt.Color;
import javax.swing.AbstractListModel;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JList;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JSlider;
import javax.swing.SwingUtilities;
import javax.swing.border.EmptyBorder;
import javax.swing.border.LineBorder;
import javax.swing.border.TitledBorder;
import javax.swing.event.ChangeEvent;
import javax.swing.event.ChangeListener;
import java.util.Calendar;
import java.util.Hashtable;
import java.util.Locale;
import java.util.Map;

public class SliList extends JFrame
                     implements ChangeListener {
    private JList<String> list;
    private JSlider years, months;
    private int year, month, currYear, currMonth, currDay;

    //private Locale locale = Locale.getDefault();
    //private Locale locale = new Locale("pl","PL");
    private Locale locale = new Locale("uk","UA");
    //private Locale locale = new Locale("el","GR");
    //private Locale locale = new Locale("ar","SA");
    //private Locale locale = new Locale("iw","IL");

    public static void main(String args[]) {
        new SliList();
    }

    SliList() {
        setDefaultCloseOperation(EXIT_ON_CLOSE);

        Calendar today = Calendar.getInstance(locale);
        currYear  = today.get(Calendar.YEAR);
        year      = currYear;
```

```
          // currMonth will be in [1,12]
        currMonth = today.get(Calendar.MONTH)+1;
        month     = currMonth;
        currDay   = today.get(Calendar.DAY_OF_MONTH);

          // this will initialize static arrays monNames
          // and dayNames in object of class CalListModel
        list = new JList<>(new CalListModel(
                           year,month,locale));

        years = new JSlider(JSlider.HORIZONTAL,
                            2000, 2025, year);
        years.setMajorTickSpacing(5);
        years.setMinorTickSpacing(1);
        years.setPaintTicks(true);
        years.setPaintLabels(true);

        years.setBorder(new TitledBorder("Years"));

        years.addChangeListener(this);
        add(years, BorderLayout.NORTH);

        Map<String,Integer> mNames =
            today.getDisplayNames(Calendar.MONTH,
                                  Calendar.SHORT,locale);
        String[] mon = new String[12];
        mNames.entrySet()
              .stream()
              .forEach(e -> mon[e.getValue()] = e.getKey());
          // Hashtable is obsolete, but necessary here...
        Hashtable<Integer,JLabel> labs = new Hashtable<>();
        for (int i = 1; i <= 12; i++) {
            JLabel lab = new JLabel(
                    CalListModel.monNames[i-1],
                    JLabel.CENTER);
            if (i==currMonth) lab.setForeground(Color.RED);
            labs.put(i, lab);
        }

        months = new JSlider(JSlider.VERTICAL,1,12,month);
          // labs must be Hashtable, not Map!
        months.setLabelTable(labs);
        months.setPaintLabels(true);
        months.setMajorTickSpacing(1);
        months.setPaintTicks(true);
        months.setSnapToTicks(true);
        months.setInverted(true);   // min value at the top
        months.setBorder(new EmptyBorder(10,10,10,10));
        months.addChangeListener(this);

        JPanel p = new JPanel(new BorderLayout());
        p.setBorder(new LineBorder(Color.blue));
        p.add(months, "East");

        // this will set max size
        list.setPrototypeCellValue("XX XXX XXXXXXXXXXXXXX");

        list.setSelectedIndex(currDay-1);
        list.setVisibleRowCount(20);

        p.add(new JScrollPane(list), BorderLayout.CENTER);
        add(p, BorderLayout.CENTER);

        SwingUtilities.invokeLater(() -> {
            pack();
            setLocationRelativeTo(null);
            setVisible(true);
        });
    }

    public void stateChanged(ChangeEvent e) {
        JSlider sl = (JSlider)e.getSource();

        if (sl.getValueIsAdjusting()) return;
        int n_year  = year,
            n_month = month;
          // which slider?
        if (sl == years)
            n_year = sl.getValue();
        else if (sl == months)
            n_month = sl.getValue();

          // if new values different than the old
        if (n_year != year || n_month != month) {
            year  = n_year;
            month = n_month;
                // new model for diffferent month/year
            list.setModel(new CalListModel(
                        year,month,locale));
            if (year==currYear && month==currMonth) {
                list.setSelectedIndex(currDay-1);
                list.ensureIndexIsVisible(currDay-1);
            } else list.ensureIndexIsVisible(0);
        }
    }
}

class CalListModel extends AbstractListModel<String> {
    private static Calendar cal = Calendar.getInstance();

    private int year;
    private int month;
    private Locale locale;

    static String[] dayNames = null;
    static String[] monNames = null;

    CalListModel(int y, int m, Locale loc) {
        year  = y;
        month = m - 1; // as months 0-based...
        locale = loc;
        cal.set(year,month,1);

        dayNames = new String[7];
        Map<String,Integer> dNames = cal.getDisplayNames(
                Calendar.DAY_OF_WEEK,Calendar.LONG,locale);
        dNames.entrySet()
              .stream()
              .forEach(e ->
                   dayNames[e.getValue()-1] = e.getKey());

        monNames = new String[12];
        Map<String,Integer> mNames = cal.getDisplayNames(
                     Calendar.MONTH,Calendar.SHORT,locale);
        mNames.entrySet()
              .stream()
              .forEach( e ->
                   monNames[e.getValue()] = e.getKey());
    }

    public String getElementAt(int i) {
          // i is 0-based, but day of month is 1-based...
        cal.set(year, month, i+1); // month is 0-based!
        int indDay = cal.get(Calendar.DAY_OF_WEEK) - 1;
        return (i+1) + " " + monNames[month] +
                       " " + cal.getDisplayName(
                                 Calendar.DAY_OF_WEEK,
                                 Calendar.LONG, locale);
    }

    public int getSize() {
        return cal.getActualMaximum(Calendar.DAY_OF_MONTH);
    }
}
```

The program displays



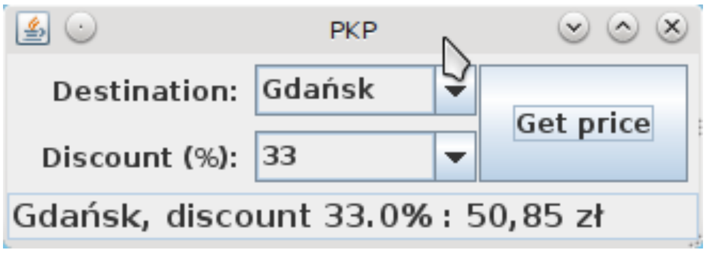
```
1 Ιουν Πέμπτη
2 Ιουν Παρασκευή
3 Ιουν Σάββατο
4 Ιουν Κυριακή
5 Ιουν Δευτέρα
6 Ιουν Τρίτη
7 Ιουν Τετάρτη
8 Ιουν Πέμπτη
9 Ιουν Παρασκευή
10 Ιουν Σάββατο
11 Ιουν Κυριακή
12 Ιουν Δευτέρα
13 Ιουν Τρίτη
14 Ιουν Τετάρτη
15 Ιουν Πέμπτη
16 Ιουν Παρασκευή
17 Ιουν Σάββατο
18 Ιουν Κυριακή
19 Ιουν Δευτέρα
20 Ιουν Τρίτη
```

In the following example, we demonstrate the use of JComboBox. It is backed by ComboBoxModel which extends ListModel. Combo boxes do not have selection models, because there is always only one item selected; instead, methods getSelectedItem and setSelectedItem have been moved to the JComboBoxModel itself.

Listing 125 PEH-Combo/MyCombo.java

```java
import java.awt.BorderLayout;
import java.awt.Font;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import javax.swing.AbstractAction;
import javax.swing.BorderFactory;
import javax.swing.JButton;
import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;
import java.text.DecimalFormat;
import java.text.NumberFormat;
import static javax.swing.SwingConstants.RIGHT;

public class MyCombo extends JFrame {
    JComboBox<City> cities;
    JComboBox<String> discount;
    JTextField  result;

    public static void main(String[] args) {
        new MyCombo();
    }

    MyCombo() {
        super("PKP");
        setDefaultCloseOperation(EXIT_ON_CLOSE);
          // to format prices
        final NumberFormat form = new DecimalFormat("#.00");
          // data
        final City[] data =
            {
                new City("Gda\u0144sk",75.90),
                new City("Krak\u00f3w",60.30),
                new City("Lublin",50.40),
                new City("Pozna\u0144",54.10),
                new City("Zabrze",61.70)
            };
          // combo with JLabels
        JLabel lcity = new JLabel("Destination:",RIGHT);
        JLabel ldisc = new JLabel("Discount (%):",RIGHT);
        cities   = new JComboBox<City>(data);
        discount = new JComboBox<String>(
            new String[]{"0", "25", "33", "50"});

        JPanel combos = new JPanel(new GridLayout(2,2,7,7));
        combos.add(lcity);
        combos.add(cities);
        combos.add(ldisc);
        combos.add(discount);

          // button "Get price"
        JButton ok =
            new JButton(new AbstractAction("Get price")
        {
            @Override
            public void actionPerformed(ActionEvent e) {
                City city =
                    (City)cities.getSelectedItem();
                double z = Double.parseDouble(
                    (String)discount.getSelectedItem());
                double c = city.getPrice();
                String price = form.format((1-z/100)*c);
                result.setText(city + ", discount " +
                        z + "% : " + price + " z\u0142");
            }
        });

        JPanel panel = new JPanel(new BorderLayout());
        panel.add(combos,BorderLayout.CENTER);
        panel.add(ok,BorderLayout.EAST);
        panel.setBorder(
            BorderFactory.createEmptyBorder(5,5,5,5));

          // field for the result
        result = new JTextField(20);
        result.setFont(new Font("Monospace",Font.BOLD,15));
        result.setEditable(false);

        setLayout(new BorderLayout());
        add(panel,BorderLayout.CENTER);
        add(result,BorderLayout.SOUTH);

        pack();
        setLocationRelativeTo(null);
        setVisible(true);
    }
}

class City implements Comparable<City> {
    private String name;
    private double price;

    City(String name, double price) {
        this.name = name;
        this.price  = price;
    }

    public double getPrice() { return price; }

    @Override
    public int compareTo(City other) {
        return name.compareTo(other.name);
    }
    @Override
    public String toString() { return name;  }
}
```

The program displays



Destination: Gdańsk

Discount (%): 33

Gdańsk, discount 33.0% : 50,85 zł

11.6 Tables

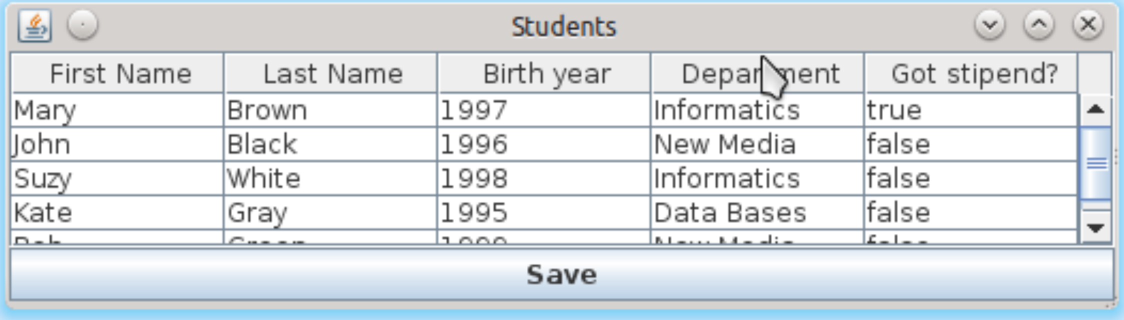
Tables (object of type **JTable**) are probably the most complicated components in Swing. In order to construct a table from our data, we need a two dimensional array of objects, rows of which will be rendered as rows of the table. Headers (names of the columns) of the table are then passed to the constructor as the second argument in the form of a one dimensional array. Instead of arrays, we also can use a **Vector**s — a little deprecated version of **ArrayList**.

Let us consider a very simple example. Here, we pass the data in the form of a two-dimensional array of **Object**s

Listing 126 PEO-SimpTable/SimpTable.java

```java
import java.awt.BorderLayout;
import java.awt.Dimension;
import java.awt.event.ActionEvent;
import javax.swing.AbstractAction;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTable;
import javax.swing.table.TableModel;

public class SimpTable extends JPanel {
    public SimpTable() {
        super(new BorderLayout());
        String[] colNames = {
            "F-name", "L-name", "Birth y.",
                      "Dept", "stipend?"
        };
        Object[][] data = {
            {"Mary", "Brown", new Integer(1997),
                     "Infor", new Boolean(true)},
            {"John", "Black", new Integer(1996),
                     "NewM",  new Boolean(false)},
            {"Suzy", "White", new Integer(1998),
                     "Infor", new Boolean(false)},
            {"Kate", "Gray",  new Integer(1995),
                     "DBas",  new Boolean(false)},
            {"Bob",  "Green", new Integer(1999),
                     "NewM",  new Boolean(false)}
        };

        JTable table = new JTable(data, colNames);
        table.setAutoCreateRowSorter(true);
        table.setPreferredScrollableViewportSize(
                             new Dimension(500, 70));
        table.setFillsViewportHeight(true);
        JScrollPane scrollPane = new JScrollPane(table);
        add(scrollPane);

        JButton b = new JButton(new AbstractAction("Show") {
            @Override
            public void actionPerformed(ActionEvent e) {
                save(table);
            }
        });
        add(b,BorderLayout.SOUTH);
    }

    private void save(JTable table) {
        TableModel m = table.getModel();
        String stars = new String(new char[68])
                      .replace('\0', '*');
        for (int r = 0; r < m.getRowCount(); ++r) {
            System.out.print("|");
            for (int c = 0; c < m.getColumnCount(); ++c) {
                Object ob = m.getValueAt(r,c);
                String claz = ob.getClass().getSimpleName();
                String val  = ob.toString();
                System.out.printf("%-" + claz.length() +
                        "s:%5s|", claz,ob.toString());
            }
            System.out.println();
        }
        System.out.println(stars);
    }

    public static void main(String... args) {
        JFrame f = new JFrame("Students");
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f.add(new SimpTable());
        f.pack();
        f.setLocationRelativeTo(null);
        f.setVisible(true);
    }
}
```

which displays



First Name	Last Name	Birth year	Department	Got stipend?
Mary	Brown	1997	Informatics	true
John	Black	1996	New Media	false
Suzy	White	1998	Informatics	false
Kate	Gray	1995	Data Bases	false

Even in this simple case, the table has quite rich functionality — one can edit the cells (after double-clicking), reorder columns, sort rows etc. However, all values are treated as just strings. Notice, that the last column is of boolean type, but still we can enter there any string whatsoever!

In order to enrich the functionality of the table, we have to use models. In fact there are three models associated with tables

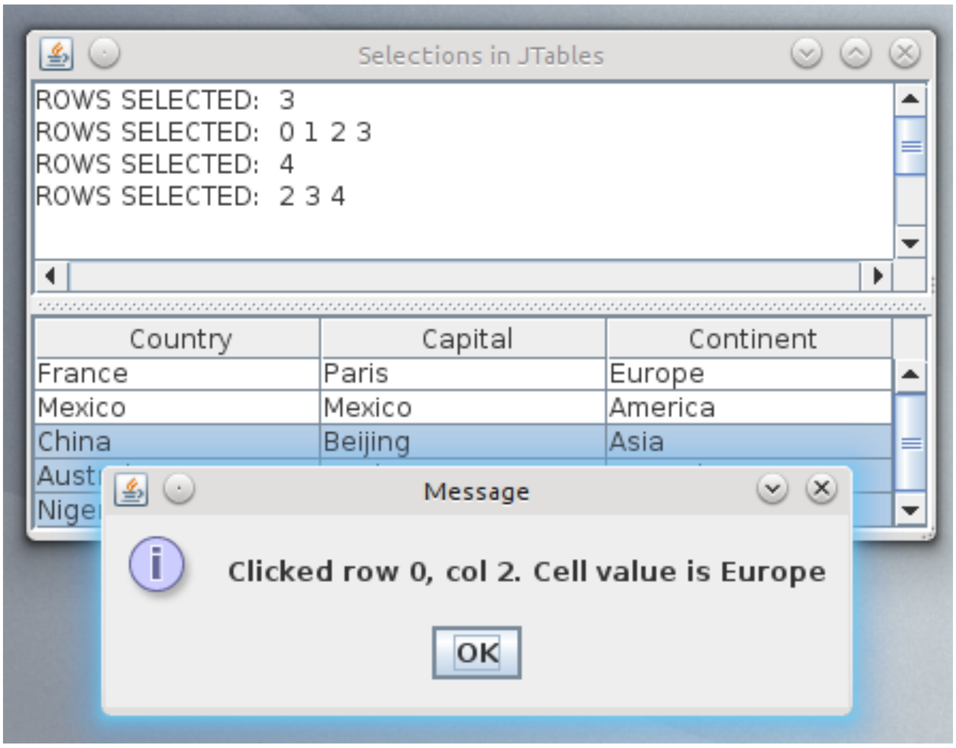
- Model of the data — an object implementing TableModel. It represents the data themselves, arranged in rows and columns. Data in one column have common type, but it may be different for different columns.
- Model describing properties of columns — an object implementing TableColumnModel and manging objects of type TableColumn. We can get the reference to such object by invoking getColumn on the model or on the table; as the argument we can pass an index of the column or its header (title). The model allows us to add, remove or reorder columns dynamically. In particular, columns are characterized by
  - header (title);
  - renderer of the header;
  - cell renderer (of type TableCellRenderer); it specifies a component representing the value of a cell (but without normal functionality of this component). By default these will be
    - JLabel with centered icon for columns of type Icon;
    - JLabel with right-aligned string for columns of numerical types (Number);
    - JCheckBox for columns of Boolean type;
    - JLabel with text obtained by invoking toString on columns of other types.
  - cell editor (of type TableCellEditor) which is represented by a 'live' component and allows us to dynamically edit values associated with cells.
- Model responsible for information on selections (as for lists) — an object implementing ListSelectionModel.

The example below demonstrates how one can attach selection and mouse listeners to a table:

Listing 127 PEP-TableSels/TableSels.java

```java
import java.awt.BorderLayout;
import java.awt.Dimension;
import java.awt.event.MouseEvent;
import java.awt.event.MouseAdapter;
import javax.swing.JFrame;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JSplitPane;
import javax.swing.JTable;
import javax.swing.JTextArea;
import javax.swing.ListSelectionModel;
import javax.swing.event.ListSelectionEvent;
import javax.swing.event.ListSelectionListener;
import static javax.swing.SwingUtilities.isRightMouseButton;

public class TableSels extends JPanel {
    public static void main(String... args) {
        JFrame f = new JFrame("Selections in JTables");
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f.add(new TableSels());
        f.pack();
        f.setLocationRelativeTo(null);
        f.setVisible(true);
    }

    public TableSels() {
        setLayout(new BorderLayout());

        JTextArea tAr = new JTextArea(10, 30);
        tAr.setEditable(false);
        JScrollPane msgPane = new JScrollPane(tAr,
                JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
                JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS);
        msgPane.setPreferredSize(new Dimension(420,100));

        String[] colNames =
                {"Country",   "Capital",  "Continent"};
        String[][] data = {
                {"France",    "Paris",    "Europe"   },
                {"Mexico",    "Mexico",   "America"  },
                {"China",     "Beijing",  "Asia"     },
                {"Australia", "Canberra", "Australia"},
                {"Nigeria",   "Abuja",    "Africa"   }};
        JTable table = new JTable(data, colNames);
        table.setPreferredScrollableViewportSize(
                          new Dimension(400, 100));
        ListSelectionModel selM = table.getSelectionModel();
        selM.addListSelectionListener(new SelListener(tAr));
        selM.setSelectionMode(
            ListSelectionModel.SINGLE_INTERVAL_SELECTION);
        table.setSelectionModel(selM);
        table.addMouseListener(new TabMouseListener());
        JScrollPane tablePane = new JScrollPane(table,
                JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
                JScrollPane.HORIZONTAL_SCROLLBAR_NEVER);
        tablePane.setPreferredSize(new Dimension(420,100));

        add(new JSplitPane(JSplitPane.VERTICAL_SPLIT,
                           msgPane, tablePane));
    }

    class SelListener implements ListSelectionListener {
        JTextArea tArea;
        SelListener(JTextArea ta) { tArea = ta; }
        @Override
        public void valueChanged(ListSelectionEvent e) {
              // to avoid printing info twice
            if (e.getValueIsAdjusting()) return;
            ListSelectionModel m =
                    (ListSelectionModel)e.getSource();
            tArea.append("ROWS SELECTED: ");
            if (m.isSelectionEmpty()) {
                tArea.append("NONE\n");
            } else {
                  // in modes 'single interval'
                  // and 'multiple interval',
                  // many rows may be selected!
                int mn = m.getMinSelectionIndex();
                int mx = m.getMaxSelectionIndex();
                for (int i = mn; i <= mx; ++i) {
                    if (m.isSelectedIndex(i)) {
                        tArea.append(" " + i);
                    }
                }
                tArea.append("\n");
            }
        }
    }

    class TabMouseListener extends MouseAdapter {
        @Override
        public void mousePressed(MouseEvent e) {
            if (!isRightMouseButton(e)) return;
            JTable table = (JTable)e.getSource();
            int row = table.rowAtPoint(e.getPoint());
            int col = table.columnAtPoint(e.getPoint());
            JOptionPane.showMessageDialog(null,
                             "Clicked row " + row + ", col " +
                             col + ". Cell value is " +
                             table.getValueAt(row,col));
                }
        }
}
```

which displays



```
ROWS SELECTED:  3
ROWS SELECTED:  0 1 2 3
ROWS SELECTED:  4
ROWS SELECTED:  2 3 4
```

Country	Capital	Continent
France	Paris	Europe
Mexico	Mexico	America
China	Beijing	Asia
Aust		
Nige		

Clicked row 0, col 2. Cell value is Europe

A more complicated example presented below demonstrates renderers — one can attach separate renderers to any column separately or to all columns of a given type, as below:

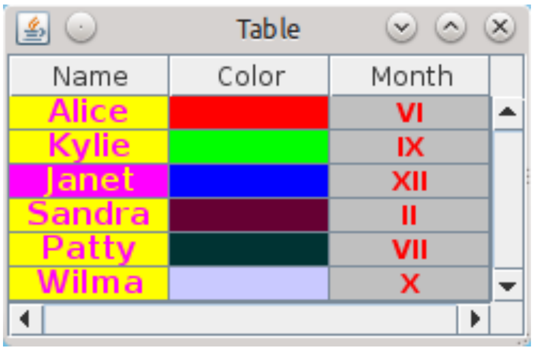
Listing 128 PET-TableEx/TableEx.java

```
import java.awt.Color;
import java.awt.Component;
import java.awt.Font;
import java.util.ArrayList;
import java.util.List;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JScrollPane;
import javax.swing.JTable;
import javax.swing.table.AbstractTableModel;
import javax.swing.table.TableCellRenderer;
import static javax.swing.JScrollPane.*;

public class TableEx {
    public static void main(String... args) {
        JTable table = new JTable(new MyTabModel());
        table.setDefaultRenderer(java.awt.Color.class,
                new MyColorCellRend());
        table.setDefaultRenderer(java.lang.Integer.class,
                new MyIntCellRend());
        table.getColumn("Name").setCellRenderer(
                new MyNameCellRend());

          // This is needed; otherwise the table
          // does not fill the whole scroll...
        table.setPreferredScrollableViewportSize(
                        table.getPreferredSize());
        JScrollPane scroll = new JScrollPane(table,
                VERTICAL_SCROLLBAR_ALWAYS,
                HORIZONTAL_SCROLLBAR_ALWAYS);
        JFrame f = new JFrame("Table");
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f.add(scroll);
        f.setLocationRelativeTo(null);
        f.pack();
        f.setVisible(true);
    }
}

class MyTabModel extends AbstractTableModel {

    ArrayList<Class>  classes = new ArrayList<>();
    ArrayList<String> colname = new ArrayList<>();

    ArrayList<String>  name = new ArrayList<>();
    ArrayList<Color>   colo = new ArrayList<>();
    ArrayList<Integer> ints = new ArrayList<>();

    ArrayList<Object>  data = new ArrayList<>();

    int ilerows;

    public static void main(String... args) {
        new MyTabModel();
    }

    private void readData() {
        String[] lines = {
            "Alice  255   0   0  6",
            "Kylie    0 255   0  9",
            "Janet    0   0 255 12",
            "Sandra 102   0  51  2",
            "Patty    0  51  51  7",
            "Wilma  201 201 255 10",
        };
        try {
```

```java
            classes.add(Class.forName("java.lang.String"));
            classes.add(Class.forName("java.awt.Color"));
            classes.add(Class.forName("java.lang.Integer"));
            colname.add("Name");
            colname.add("Color");
            colname.add("Month");
        } catch(ClassNotFoundException e) {
            e.printStackTrace();
            System.exit(1);
        }
        ilerows = lines.length;
        for (String line : lines) {
            String[] fields = line.split("\\s+");
            name.add(fields[0]);
            colo.add(new Color(Integer.parseInt(fields[1]),
                               Integer.parseInt(fields[2]),
                               Integer.parseInt(fields[3])));
            ints.add(Integer.valueOf(fields[4]));
        }
        data.add(name);
        data.add(colo);
        data.add(ints);
    }

    public MyTabModel() {
        readData();
    }

    @Override
    public int getColumnCount() {
        return data.size();
    }

    @Override
    public Object getValueAt(int r, int c) {
        return ((List)data.get(c)).get(r);
    }

    @Override
    public String getColumnName(int c) {
        return colname.get(c);
    }

    @Override
    public int getRowCount() {
        return ilerows;
    }

    @Override
    public Class<?> getColumnClass(int c) {
        return classes.get(c);
    }

    @Override
    public boolean isCellEditable(int r, int c) {
        return false;
    }
}

class MyColorCellRend extends JLabel
                      implements TableCellRenderer {
    public MyColorCellRend() {
        setOpaque(true);
    }

    public Component
    getTableCellRendererComponent(
            JTable table, Object color, boolean isSelected,
            boolean hasFocus, int row, int column) {
        setBackground((Color)color);
        return this;
    }
}

class MyIntCellRend extends JLabel
                    implements TableCellRenderer {
    static final String[] romans = {
        "?", "I", "II", "III", "IV", "V", "VI",
        "VII", "VIII", "IX", "X", "XI", "XII"
    };

    public MyIntCellRend() {
        setOpaque(true);
        setBackground(Color.LIGHT_GRAY);
        setForeground(Color.RED);
        setHorizontalAlignment(JLabel.CENTER);
    }

    public Component
    getTableCellRendererComponent(
            JTable table, Object month, boolean isSelected,
            boolean hasFocus, int row, int column) {
        setText(romans[(Integer)month]);
        return this;
    }
}

class MyNameCellRend extends JLabel
                     implements TableCellRenderer {
    public MyNameCellRend() {
```

```java
        setOpaque(true);
        setHorizontalAlignment(JLabel.CENTER);
        setFont(new Font("Sansserif", Font.BOLD, 15));
    }

    public Component
    getTableCellRendererComponent(
            JTable table, Object str, boolean isSelected,
            boolean hasFocus, int row, int column) {
        if (isSelected) {
            setBackground(Color.MAGENTA);
            setForeground(Color.YELLOW);
        } else {
            setForeground(Color.MAGENTA);
            setBackground(Color.YELLOW);
        }
        setText(str.toString());
        return this;
    }
}
```

which displays



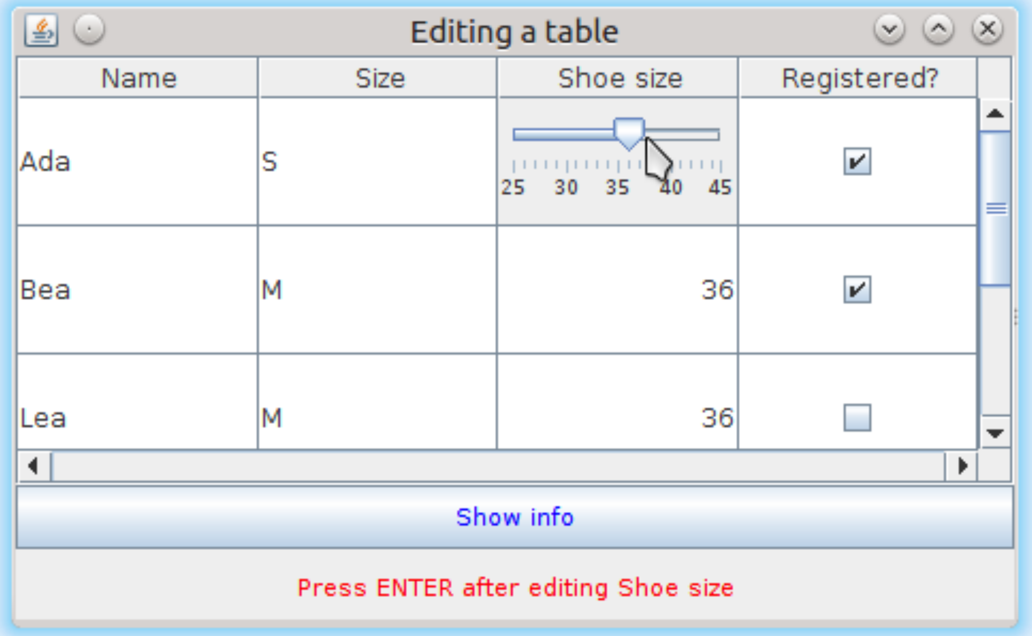
Name	Color	Month
Alice	🟥	VI
Kylie	🟩	IX
Janet	🟦	XII
Sandra	🟫	II
Patty	⬛	VII
Wilma	⬜	X

In the example below, we demonstrate how to edit cells of a table in a non-standard way: the table contains columns which allow to edit the values using JCheckBoxes (boolean values, functionality provided by the library) or JSliders (Integers) and JComboBoxes (Strings) — the latter two by custom editors:

Listing 129 PES-TableEdit/TableEdit.java

```java
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Component;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.GridLayout;
import javax.swing.AbstractCellEditor;
import javax.swing.DefaultCellEditor;
import javax.swing.JButton;
import javax.swing.JCheckBox;
import javax.swing.JComboBox;
import javax.swing.JComponent;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JSlider;
import javax.swing.JTable;
import javax.swing.table.AbstractTableModel;
import javax.swing.table.TableCellEditor;
import javax.swing.table.TableModel;
import static javax.swing.JScrollPane.*;

public class TableEdit extends JFrame {
    static final JButton INF = new JButton("Show info");
    static final JLabel  MSG = new JLabel(
            "Except the first, columns are editable!",
                                    JLabel.CENTER);
    static {
        for (JComponent c : new JComponent[]{INF,MSG}) {
            c.setForeground(Color.RED);
            c.setOpaque(true);
            c.setFont(new Font("Dialog",Font.PLAIN,11));
            c.setMinimumSize(new Dimension(50,30));
            c.setPreferredSize(new Dimension(100,30));
        }
        INF.setForeground(Color.BLUE);
    }

    TableModel model = new MyTableModel();

    public static void main (String[] args) {
        new TableEdit();
    }
    TableEdit() {
        super("Editing a table");
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        JTable table = new JTable(model);
        table.getColumn("Shoe size")
             .setCellEditor(new MySliderEditor());
        table.getColumn("Size")
             .setCellEditor(new MyComboEditor());
        table.setRowHeight(60);
        table.setRowSelectionAllowed(false);
        JScrollPane scroll = new JScrollPane(table,
                JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
                JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS);
        add(scroll);
        JPanel p = new JPanel(new GridLayout(2,1,3,3));
        INF.addActionListener(e -> showData(model));
        p.add(INF);
        p.add(MSG);
        add(p,BorderLayout.SOUTH);
        pack();
        setLocationRelativeTo(null);
        setVisible(true);
    }
    static void showData(TableModel m) {
        String stars = new String(new char[71])
                      .replace('\0', '*');
        for (int r = 0; r < m.getRowCount(); ++r) {
            System.out.print("|");
            for (int c = 0; c < m.getColumnCount(); ++c) {
                Object ob = m.getValueAt(r,c);
                String claz = ob.getClass().getSimpleName();
                String val  = ob.toString();
                System.out.printf(" %-" + claz.length() +
                        "s : %5s |", claz,ob.toString());
            }
            System.out.println();
        }
        System.out.println(stars);
    }
}

class MyTableModel extends AbstractTableModel {
    static String[] headers = {
            "Name", "Size", "Shoe size", "Registered?" };
    static String[] sizes = {"S","M","L","XL","XXL"};
    static Integer minShoeSize = 25, maxShoeSize = 45;

    Object[][] data = {
        {"Ada",sizes[0],new Integer(34),Boolean.TRUE},
        {"Bea",sizes[1],new Integer(36),Boolean.TRUE},
        {"Lea",sizes[1],new Integer(36),Boolean.FALSE},
        {"Kim",sizes[3],new Integer(40),Boolean.TRUE},
        {"Zoe",sizes[2],new Integer(38),Boolean.FALSE}};
      // 3 methods below are abstract
      // and have to be overridden!
    @Override
    public int getColumnCount() { return headers.length; }
    @Override
    public int getRowCount()    { return data.length;    }
    @Override
    public Object getValueAt(int r, int c) {
        return data[r][c];
    }

    @Override
    public String getColumnName(int c) {return headers[c];}
    @Override
    public Class<?> getColumnClass(int c) {
        return getValueAt(0, c).getClass();
    }
    @Override
    public boolean isCellEditable(int r, int c) {
        return c > 0;
    }
    @Override
    public void setValueAt(Object val, int r, int c) {
        data[r][c] = val;
        fireTableCellUpdated(r,c);
    }
}

class MySliderEditor extends AbstractCellEditor
                     implements TableCellEditor {
    private static final Font font =
                            new Font("Dialog",Font.PLAIN,9);
    private JPanel  panel = new JPanel(new BorderLayout());
    private JSlider slider;
    private Integer value;
    MySliderEditor() {
        slider = new JSlider(
                    JSlider.HORIZONTAL,
                    MyTableModel.minShoeSize,
                    MyTableModel.maxShoeSize,
                   (MyTableModel.minShoeSize +
                    MyTableModel.maxShoeSize)/2);
        slider.setMajorTickSpacing(5);
        slider.setMinorTickSpacing(1);
        slider.setPaintTicks(true);
        slider.setPaintLabels(true);
        slider.setFont(font);

        slider.addChangeListener(e -> {
                if (slider.getValueIsAdjusting()) return;
                value = slider.getValue();
             }
        );
        panel.add(slider);
    }
    @Override
    public Component getTableCellEditorComponent(
                JTable table, Object v, boolean isSelected,
                int row, int column) {
        value = (Integer)v;
        return panel;
    }
    @Override
    public Object getCellEditorValue() {
      return value;
    }
}

class MyComboEditor extends DefaultCellEditor {
    private JPanel  panel = new JPanel();
    private JComboBox<String> combo;
    private String value;
    MyComboEditor() {
        super(new JCheckBox());
        combo = new JComboBox<>(MyTableModel.sizes);
        combo.setMaximumSize(new Dimension(100,20));
        combo.setPreferredSize(new Dimension(100,20));
        combo.setSelectedIndex(1);
        combo.addActionListener(e ->  {
            value = (String)combo.getSelectedItem();
            fireEditingStopped();
        });
        panel.add(combo);
    }
    @Override
    public Component getTableCellEditorComponent(
                JTable table, Object v, boolean isSelected,
                int row, int column) {
        value = (String)v;
        return panel;
    }
    @Override
    public Object getCellEditorValue() {
      return value;
    }
}
```

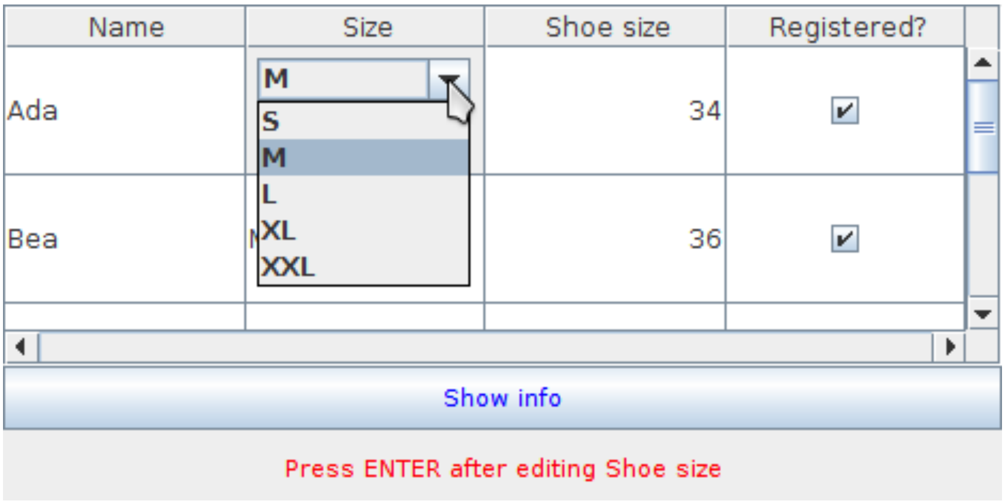
The third column (of type **Integer**) can be edited using a **JSlider** — the user signals the end of selecting a new value by pressing ENTER



Name	Size	Shoe size	Registered?
Ada	S		☑
Bea	M	36	☑
Lea	M	36	□

Press ENTER after editing Shoe size

The second column (of type **String**) is edited with a **JComboBox** which allows to select one of a few predefined options



Name	Size	Shoe size	Registered?
Ada	M	34	☑
Bea	M	36	☑

Press ENTER after editing Shoe size

Section 12

JavaFX — introduction

12.1 Introduction

Java provides relatively simple tools that can be used to build rich graphical user interfaces; they are collected in a set of packages that collectively are called JavaFX. Main classes of JavaFx applications should extend **Application** from ***javafx.application*** package. Such classes are treated in a special way: when loaded, the JVM creates an object of the main class, calls **init** on it, and then the **start** function on a separate thread — the so called **JavaFX Application Thread** (or just **application thread**, for short), which corresponds to Event Dispatch Thread in Swing. This means that no **main** function is needed! However, most IDE's require every Java application to start from **main**, so we usually add this function with a trivial implementation

```
public class OurMainClass extends Application {
    // ...
    public static void main(String[] args) {
        launch(args);
    }
    // ...
}
```

or, to emphasise the fact that **launch** is static,

```
            Application.launch(args);
```

When the application is being closed, another special method is invoked — **stop** (still on the application thread).

Let us look at an example. To get a specific layout, we use here a **StackPane** with a button (**Button**); we also attach an action to the button. To do this, we used a lambda, but

```
btn.setOnAction(
    new EventHandler<ActionEvent>() {
        @Override
        public void handle(ActionEvent event) {
            System.out.println("Hello!");
        }
    }
);
```

or

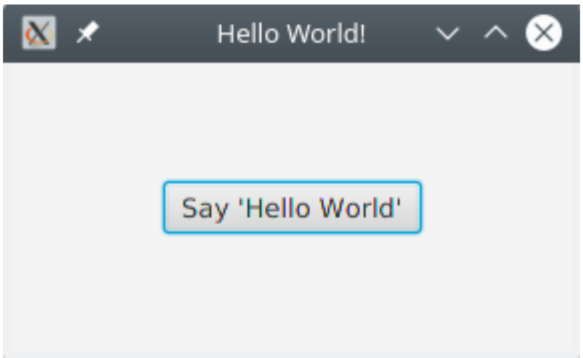
```
btn.addEventHandler(
    ActionEvent.ACTION,
    e -> System.out.println("Hello!"));
```

would also work (as we will explain in Sec. 12.3):

Listing 130 JXA-First/First.java

```java
package first;

import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class First extends Application {
    public static void main(String[] args) {
        System.out.println("Entering main() on " +
                           Thread.currentThread().getName());
        launch(args);
    }

    @Override
    public void init() {
        System.out.println("Entering init() on " +
                           Thread.currentThread().getName());
    }

    @Override
    public void stop() {
        System.out.println("Entering stop() on " +
                           Thread.currentThread().getName());
    }

    @Override
    public void start(Stage stage) {
        System.out.println("Entering start() on " +
                           Thread.currentThread().getName());
        Button btn = new Button("Say 'Hello'");
        btn.setOnAction(e ->System.out.println("Hello!"));

        StackPane root = new StackPane();
        root.getChildren().add(btn);

        Scene scene = new Scene(root, 300, 250);

        stage.setTitle("Hello World!");
        stage.setScene(scene);
        stage.show();
        System.out.println("Leaving start()");
    }
}
```

The programs displays:



and prints

```
Entering main() on main
Entering init() on JavaFX-Launcher
Entering start() on JavaFX Application Thread
Leaving start()
Hello World!
Entering stop() on JavaFX Application Thread
```

As we can see, before entering the application proper, the **init** method is invoked (on a separate thread called JavaFX-Launcher), so one can put there a code which, for example, prepares some data, connects to a data base, etc.

Then, on the application thread, **start** method is called. As an argument, it gets an object of class **Stage**. This roughly corresponds to the heavy-weight **JFrame** object — it depends on the platform in use (desktop, mobile, etc).

By using **setScene** method, we specify a 'scene', i.e., a content of the stage (roughly corresponding to the contentPane of a **JFrame**). The scene in turn contains a component which will be the root of the whole hierarchy of components. This object is traditionally called root and it can be any object inheriting from the abstract class **Parent**. This class in turn inherits from **Node** which represents all graphical components. Nodes usually can have other nodes as their 'children' (they are then called 'branches'), but some (called 'leaves') cannot (**Rectangle** and other **Shape**s, **Text**, etc.). Children can be added to a **Node** by method **add** invoked on the **ObservableList<Node>** associated with the given **Node** — it can be obtained by calling **getChildren** on the node. The **add** method adds one child, while **addAll** method adds an arbitrary number of children.

It follows that the whole structure of the JavaFX GUI has the form of a tree rooted at the node passed to the scene.

12.2 Layouts

Usually, the root component is a layout (from the *javafx.scene.layout* package). The basic one is **Pane**, from which other layouts inherit: **AnchorPane**, **BorderPane**, **FlowPane**, **GridPane**, **HBox**, **VBox**, **StackPane**, **TilePane** and others.

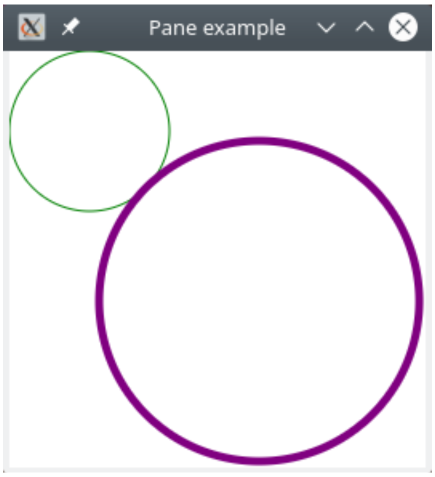
As we can see, the approach here is different than it was in Swing: in Swing we can create a component (like **JPanel**) and then set its layout; in JavaFX we create various components that already have a predetermined layout.

The simplest layout is provided by **Pane**. All its children are located at (0,0) position, so to avoid overlapping we have to specify their location manually or set appropriate padding. In the example below the c1 circle is given coordinates of the center (passed to the constructor) and it will be, by default, located in such a way that the square circumscribed on it will be touching the upper-left corner. The second **Circle**, c2, is relocated after creation to be tangent to the first one:

Listing 131 JXD-PlainPane/PlainPane.java

```java
package pane;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.stage.Stage;

public class PlainPane extends Application {
    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage stage) {
        Pane root = new Pane();
        double r = 50, d = r*(1+3/Math.sqrt(2));

        Circle c1 = new Circle(r, r, r);
        c1.setFill(Color.TRANSPARENT);
        c1.setStroke(Color.GREEN);

        Circle c2 = new Circle(d, d, 2*r);
          // or
        // Circle c2 = new Circle(2*r);
        // c2.relocate(d-2*r, d-2*r);

        c2.setFill(Color.TRANSPARENT);
        c2.setStroke(Color.PURPLE);
        c2.setStrokeWidth(5);

        root.getChildren().addAll(c1, c2);
        stage.setScene(new Scene(root));
        stage.setTitle("Pane example");
        stage.show();
    }
}
```

The program displays



Note that here we haven't specified the sizes of the Scene — the Pane layout will calculate them so they are as small as possible to contain all the children.

There is a similar layout, StackPane, which also superimposes its children but not at the (0,0) location but in the center.

The next example demonstrates the BorderPane class. It's roughly like a JPanel from Swing with BorderLayout layout. However, in JavaFX the five areas of the layout are called LEFT, RIGHT, TOP, BOTTOM and CENTER; we add components to these areas by using special methods, like setRight etc. The example illustrates also components with other layouts: HBox, VBox and TextArea. There are also the so called controls here (object of classes extending the Control class). The Control objects represent models and contain methods which manipulate the data. The corresponding 'views' are implemented by classes extending Skin. Normally, we don't have to deal with skins, as reasonable defaults are provided. The skin calculates sizes and is responsible for detecting relevant events occurring on the control.

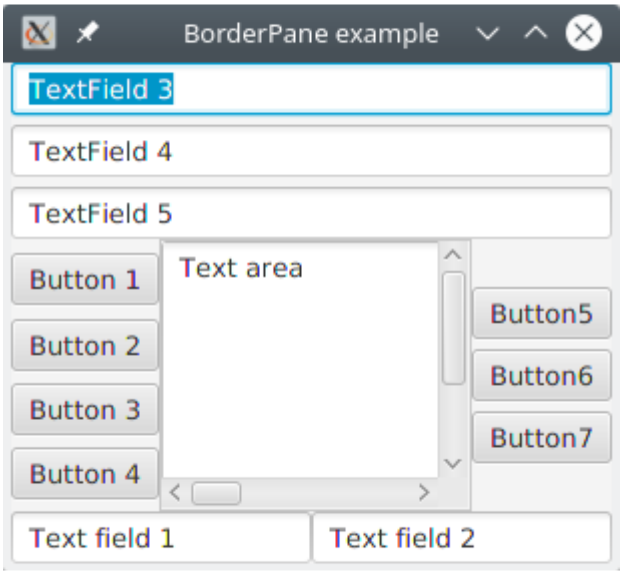
Important examples of controls, some of which are demonstrated in the example, are Button, CheckBox, Label, Menu, MenuItem, MenuBar, ScrollPane, TextArea, TextField, Slider, Spinner, and many others.

Listing 132 JXG-BorderPane/BorderPaneEx.java

```java
package borderpane;

import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Node;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.ScrollPane;
import javafx.scene.control.TextArea;
import javafx.scene.control.TextField;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.HBox;
import javafx.scene.layout.Priority;
import javafx.scene.layout.Region;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;
```

```java

public class BorderPaneEx extends Application {

    @Override
    public void start(Stage stage) {

        // left column of buttons
        VBox leftVBox = new VBox();
        leftVBox.getChildren().add(getSpace());
        for (int i = 1; i <=4; ++i) {
          leftVBox.getChildren().addAll(
              new Button("Button " + i), getSpace());
        }

        // right column of buttons
        VBox rightVBox = new VBox(5);
        rightVBox.setAlignment(Pos.CENTER);
        for (int i = 5; i <=7; ++i)
          rightVBox.getChildren().add(
              new Button("Button" + i));

        // bottom row of two text fields
        HBox bottomHBox = new HBox();
        TextField tf1 = new TextField("Text field 1");
        TextField tf2 = new TextField("Text field 2");
        HBox.setHgrow(tf1, Priority.ALWAYS);
        HBox.setHgrow(tf2, Priority.ALWAYS);
        bottomHBox.getChildren().addAll(tf1, tf2);

        // top column of three text fields
        VBox topVBox = new VBox(5);
        for (int i = 3; i <=5; ++i)
          topVBox.getChildren().add(
              new TextField("TextField " + i));

        // center
        TextArea ta = new TextArea("Text area");
        ScrollPane scroll = new ScrollPane(ta);
        scroll.setHbarPolicy(
            ScrollPane.ScrollBarPolicy.ALWAYS);
        scroll.setVbarPolicy(
            ScrollPane.ScrollBarPolicy.ALWAYS);

        BorderPane root = new BorderPane();
        root.setTop(topVBox);
        root.setCenter(scroll);
        root.setBottom(bottomHBox);
        root.setLeft(leftVBox);
        root.setRight(rightVBox);

        Scene scene = new Scene(root, 300, 250);

        stage.setTitle("BorderPane example");
        stage.setScene(scene);
        stage.show();
    }

    private Node getSpace() {
        Region space = new Region();
        VBox.setVgrow(space, Priority.ALWAYS);
        return space;
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

The program displays



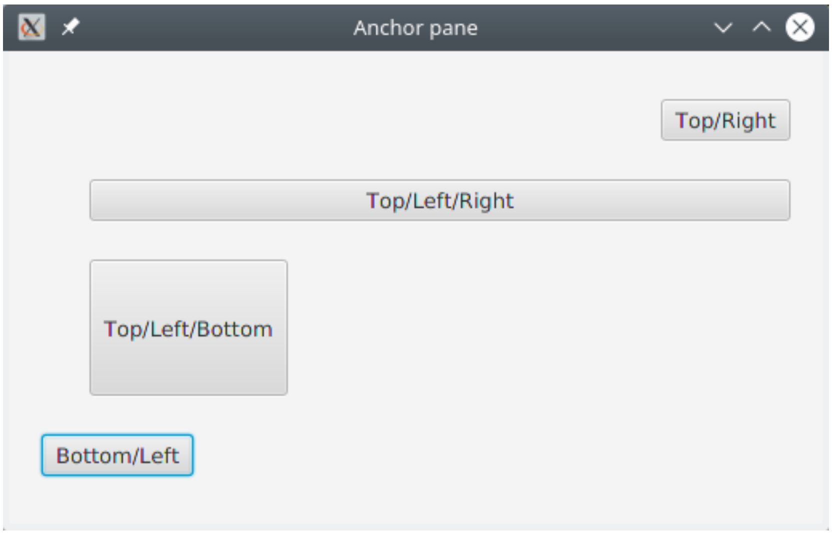
The **AnchorPane** layout allows to specify distances of components from the edges of the enclosing node; for example

Listing 133 JXB-Anchors/Anchors.java

```
package anchors;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.stage.Stage;
import javafx.scene.control.Button;
import javafx.scene.layout.AnchorPane;

public class Anchors extends Application {
```

```java

    @Override
    public void start(Stage stage) throws Exception {
        AnchorPane root = new AnchorPane();

        Button b1 = new Button("Bottom/Left");
        AnchorPane.setBottomAnchor(b1, 30.);
        AnchorPane.setLeftAnchor(b1, 20.);

        Button b2 = new Button("Top/Right");
        AnchorPane.setTopAnchor(b2, 30.);
        AnchorPane.setRightAnchor(b2, 20.);

        Button b3 = new Button("Top/Left/Right");
        AnchorPane.setTopAnchor(b3, 80.);
        AnchorPane.setLeftAnchor(b3, 50.);
        AnchorPane.setRightAnchor(b3, 20.);

        Button b4 = new Button("Top/Left/Bottom");
        AnchorPane.setTopAnchor(b4, 130.);
        AnchorPane.setLeftAnchor(b4, 50.);
        AnchorPane.setBottomAnchor(b4, 80.);

        root.getChildren().addAll(b1, b2, b3, b4);
        Scene scene = new Scene(root, 250, 300);
        stage.setTitle("Anchor pane");
        stage.setScene(scene);
        stage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

displays



Probably the most versatile layout is provided by **GridPane**. The layout consists of a grid of cells where we can put children. Each child may occupy any rectangular subarea of adjacent cells. Adding children to the layout, we specify indices of the upper-left cell and horizontal and vertical span (number of rows and columns that the child will occupy); the default span is 1. We don't need to specify the total number of rows and columns in advance — they will be calculated automatically. Some cells may remain empty. All cells in one column will have the same width and all cells in one row will have the same height, but otherwise the sizes of cells do not have to be the same (there is another layout, **TilePane**, where all cells do have the same size).

The example also demonstrates how nodes can be rotated (lines 25-26), setting padding, gaps and alignments (lines 54-62) and the use of constraints (lines 64-68; there are much more options that could have been used here — see the documentation).

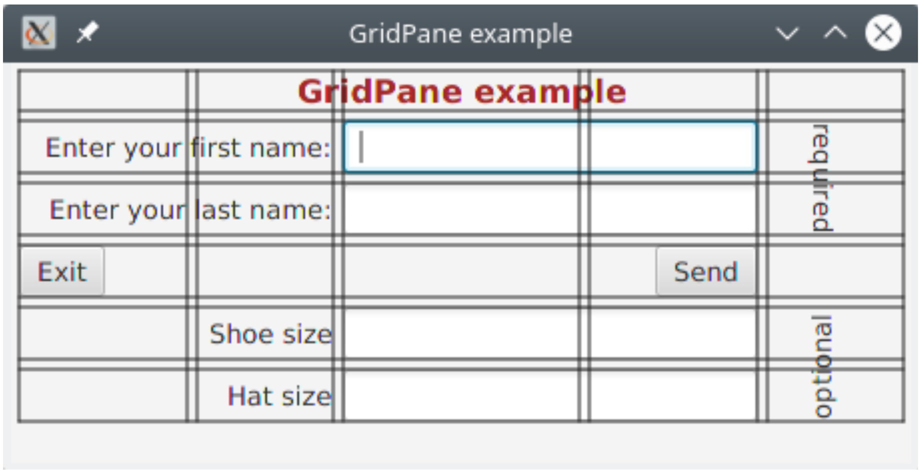
Listing 134 JXI-Grid/GridPaneEx.java

```
 1  package gridpane;
 2
 3  import javafx.application.Application;
 4  import javafx.geometry.HPos;
 5  import javafx.geometry.Insets;
 6  import javafx.scene.Scene;
 7  import javafx.scene.control.Button;
 8  import javafx.scene.control.Label;
 9  import javafx.scene.control.TextField;
10  import javafx.scene.layout.ColumnConstraints;
11  import javafx.scene.layout.GridPane;
12  import javafx.scene.paint.Color;
13  import javafx.scene.text.Font;
14  import javafx.scene.text.FontWeight;
15  import javafx.stage.Stage;
16
17  public class GridPaneEx extends Application {
18
19      @Override
20      public void start(Stage primaryStage) {
        Button send  = new Button("Send");
        Button exit  = new Button("Exit");
        Label  req   = new Label("required");
        Label  opt   = new Label("optional");
        req.setRotate(90);
        opt.setRotate(270);
        Label  text  = new Label("GridPane example");
        text.setFont(Font.font("System", FontWeight.BOLD, 16));
        text.setTextFill(Color.BROWN);
        Label  first = new Label("Enter your first name:");
        Label  last  = new Label("Enter your last name:");
        Label  shoe  = new Label("Shoe size");
        Label  hat   = new Label("Hat size");
        TextField fn = new TextField();
        TextField ln = new TextField();
        TextField ss = new TextField();
        TextField hs = new TextField();

        GridPane root = new GridPane();
        root.add(text,  0, 0, 5, 1);
        root.add(first, 0, 1, 2, 1);
        root.add(last,  0, 2, 2, 1);
        root.add(fn,    2, 1, 2, 1);
        root.add(ln,    2, 2, 2, 1);
        root.add(exit,  0, 3);
        root.add(send,  3, 3);
        root.add(shoe,  0, 4, 2, 1);
        root.add(hat,   0, 5, 2, 1);
        root.add(ss,    2, 4, 2, 1);
        root.add(hs,    2, 5, 2, 1);
        root.add(req,   4, 1, 1, 2);
        root.add(opt,   4, 4, 1, 2);

        root.setPadding(new Insets(4));
        root.setHgap(5);
        root.setVgap(5);
        GridPane.setHalignment(text,  HPos.CENTER);
        GridPane.setHalignment(first, HPos.RIGHT);
        GridPane.setHalignment(last,  HPos.RIGHT);
        GridPane.setHalignment(shoe,  HPos.RIGHT);
        GridPane.setHalignment(hat,   HPos.RIGHT);
        GridPane.setHalignment(send,  HPos.RIGHT);
        int[] percWidth = {20, 16, 28, 20, 16};
        for (int col = 0; col < 5; ++col) {
            ColumnConstraints c = new ColumnConstraints();
            c.setPercentWidth(percWidth[col]);
            root.getColumnConstraints().add(c);
        }

        //root.setGridLinesVisible(true);
```

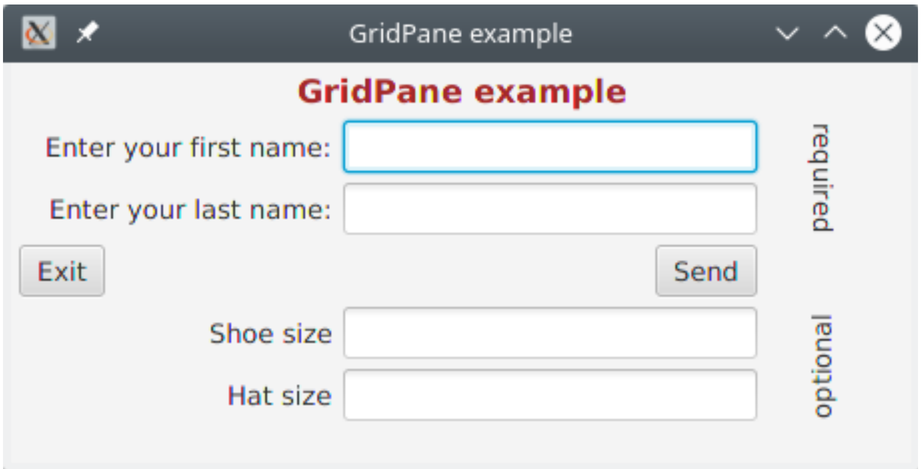
```
71
72          Scene scene = new Scene(root, 450, 200);
73          primaryStage.setTitle("GridPane example");
74          primaryStage.setScene(scene);
75          primaryStage.show();
76      }
77
78      public static void main(String[] args) {
79          launch(args);
80      }
81  }
```

Note also the method **setGridLinesVisible** (line 70). Passing **true** we can see the grid lines what is very useful when designing the GUI.

The program displays, with grid lines visible



and without them



12.3 Events

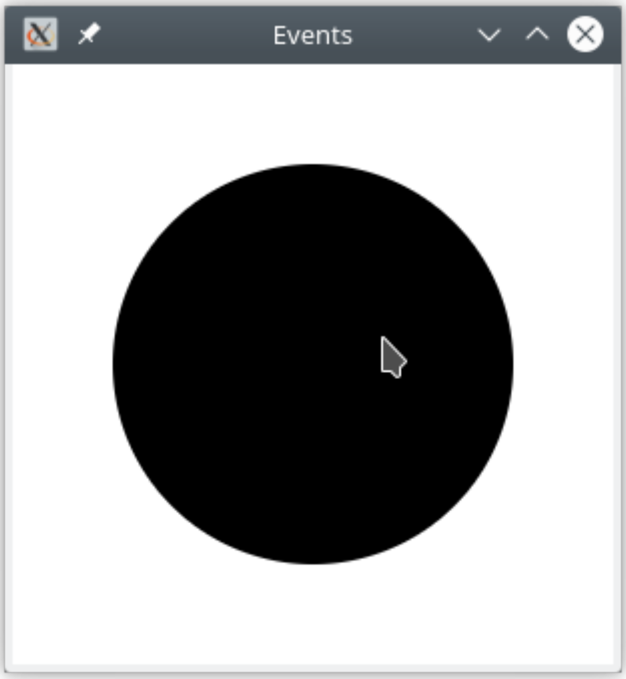
Events notify the system that something has happened. This ‘something’ may be a click on a button, pressing a key, mouse movement, etc. Events, represented as objects of type **javafx.event.Event** and its many subclasses, have properties *source*, *target* and *type*. The target is the the component on which the event occurred (a button, a text field,…) while the source is initially its farther ancestor (usually the stage). When an event is handled, it is delivered to all nodes from the source to the target along the so called *event dispatch chain* — this is the *capturing phase*. Then it goes back to the original source (*bubbling phase*). At each step the current node becomes the source, while the target remains the same. Every component along the chain may have a registered handlers (**EventHandler**) with one method, **handle**, receiving the event): handlers registered with **addEventFilter** will be triggered at the capturing phase, while those registered with **addEventHandler** – at the bubbling phase. Handlers may 'consume' events, and then processing stops.

The following program

Listing 135 JYL-EventPath/EventPath.java

```java
package eventpath;

import javafx.application.Application;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.input.MouseEvent;
import javafx.scene.layout.StackPane;
import javafx.scene.shape.Circle;
import javafx.stage.Stage;

public class EventPath extends Application {
    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage stage) {
        EventHandler<MouseEvent> handler =  e -> {
            System.out.println(
                "Handler - " + e.getEventType() + "\n" +
                "S:" + e.getSource().getClass() + "\n" +
                "T:" + e.getTarget().getClass() + "\n");
        };
        EventHandler<MouseEvent> filter =  e -> {
            System.out.println(
                "Filter  - " + e.getEventType() + "\n" +
                "S:" + e.getSource().getClass() + "\n" +
                "T:" + e.getTarget().getClass() + "\n");
        };
        stage.addEventHandler(MouseEvent.MOUSE_PRESSED,handler);
        stage.addEventFilter (MouseEvent.MOUSE_PRESSED,filter);

        StackPane root = new StackPane();
        root.addEventHandler(MouseEvent.MOUSE_PRESSED, handler);
        root.addEventFilter (MouseEvent.MOUSE_PRESSED, filter);

        Circle circ = new Circle(100);
        circ.addEventHandler(MouseEvent.MOUSE_PRESSED, handler);
        circ.addEventFilter (MouseEvent.MOUSE_PRESSED, filter);

        root.getChildren().add(circ);

        stage.setScene(new Scene(root, 300, 300));
        stage.setTitle("Events");
        stage.show();
    }
}
```

displays a simple GUI with a cirle in a **StackPane** which in turn is put into the **Stage** object:



and prints, after clicking on the circle,

```
Filter  - MOUSE_PRESSED
S:class javafx.stage.Stage
T:class javafx.scene.shape.Circle

Filter  - MOUSE_PRESSED
S:class javafx.scene.layout.StackPane
T:class javafx.scene.shape.Circle

Filter  - MOUSE_PRESSED
S:class javafx.scene.shape.Circle
T:class javafx.scene.shape.Circle
```

```
Handler - MOUSE_PRESSED
S:class javafx.scene.shape.Circle
T:class javafx.scene.shape.Circle

Handler - MOUSE_PRESSED
S:class javafx.scene.layout.StackPane
T:class javafx.scene.shape.Circle

Handler - MOUSE_PRESSED
S:class javafx.stage.Stage
T:class javafx.scene.shape.Circle
```

what illustrates the route from the original source (the stage) to the target (the circle) and back.

Of course, usually we are only interested in handling events at the target, ignoring other nodes along the chain.

There are many types of events — they form a hierarchy rooted at **Event** with subtypes **InputEvent**, **ActionEvent**, **WidowEvent** and many others. Input events in turn can be of type **KeyEvent**, **MouseEvent**, and others.

Key events are further extended to the more specific **KEY_PRESSED**, **KEY_RELEASED** and **KEY_TYPED** (**KeyEvent.ANY** stands for any of them).
Most specific mouse events are **MOUSE_ENTERED**, **MOUSE_EXITED**, **MOUSE_PRESSED**, **MOUSE_RELEASED**, **MOUSE_MOVED**, **MOUSE_CLICKED** an a few others.

Handlers (listeners in the Swing parlance) are registered with nodes by the method

addEventHandler(EventType, EventHandler)

which takes a type of events we are interested in and an object implementing the **EventHandler** interface. The interface declares one method, **handle(event)**, so it is a functional interface and may be implemented as a lambda.

For many types of events there exist convenience methods, where the type of events that are to be handled is already indicated by the very name of a method, so their single argument is just an appropriate handler:

setOnAction(handler)
setOnKeyPressed(handler)
setOnKeyReleased(handler)
setOnKeyTyped(handler)
setOnMousePressed(handler)
setOnMouseReleased(handler)

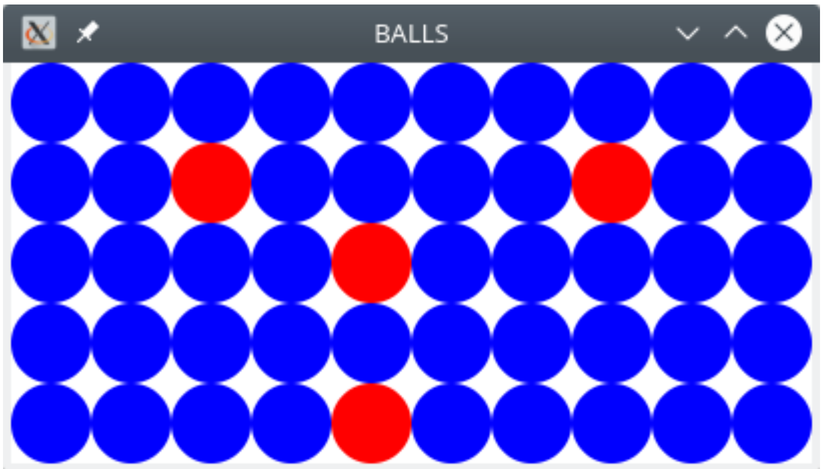
and so on.

Examples of event handling can be found in the following program

Listing 136 JXK-Events/Balls.java

```java
package balls;

import javafx.application.Application;
import javafx.event.EventHandler;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.input.KeyCode;
import javafx.scene.input.KeyEvent;
import javafx.scene.input.MouseEvent;
import javafx.scene.layout.Background;
import javafx.scene.layout.BackgroundFill;
import javafx.scene.layout.CornerRadii;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.stage.Stage;

public class Balls extends Application {

    private static final int NROWS  =  5;
    private static final int NCOLS  = 10;
    private static final int R      = 20;

    @Override
    public void start(Stage stage) {
        stage.setTitle("BALLS");
        Pane root = new Pane();
        Background bgBlack = new Background(
            new BackgroundFill(
                            // or null           // or null
                Color.BLACK,CornerRadii.EMPTY,Insets.EMPTY));
        Background bgWhite = new Background(
            new BackgroundFill(
                Color.WHITE,CornerRadii.EMPTY,Insets.EMPTY));
        root.setBackground(bgWhite);

        stage.addEventHandler(KeyEvent.KEY_PRESSED,
            new EventHandler<KeyEvent>() {
                public void handle(KeyEvent e) {
                    if (e.getCode() == KeyCode.W)
                        root.setBackground(bgWhite);
                    else if (e.getCode() == KeyCode.B)
                        root.setBackground(bgBlack);
                }
            }
        );
          // using lambda
        EventHandler<MouseEvent> handler = e -> {
            Circle c = (Circle)e.getSource();
            c.setFill(c.getFill() == Color.BLUE ?
                          Color.RED : Color.BLUE);
        };

        for (int r = 0; r < NROWS; ++r) {
            for (int c = 0; c < NCOLS; ++c) {
                Circle circle =
                        new Circle(R + c*2*R, R + r*2*R, R);
                circle.setFill(Color.BLUE);
                  // or circle.setOnMouseClicked(handler);
                circle.addEventHandler(
                        MouseEvent.MOUSE_PRESSED, handler);
                root.getChildren().add(circle);
            }
        }
        stage.setScene(new Scene(root,2*R*NCOLS,2*R*NROWS));
        stage.show();
    }


    public static void main(String[] args) {
        launch(args);
    }
}
```

which produces



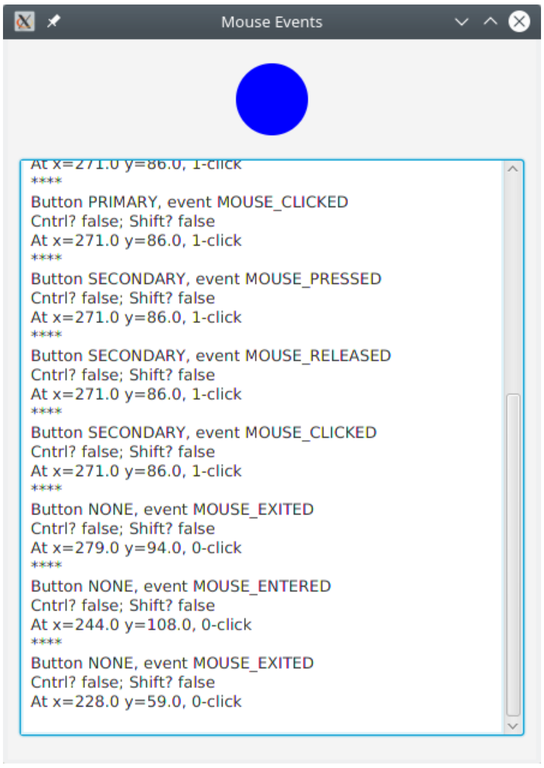
Handling mouse events is further illustrated in the following example

Listing 137 JXJ-MouseEvents/MouseFX.java

```
package mouse;

import javafx.application.Application;
import javafx.event.EventHandler;
import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.TextArea;
import javafx.scene.layout.VBox;
import javafx.scene.input.MouseEvent;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
```

```java
import javafx.stage.Stage;

public class MouseFX extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    TextArea ta = null;

    @Override
    public void start(Stage stage) {
        ta = new TextArea();
        ta.setPrefRowCount(30);
        ta.setPrefColumnCount(30);
        ta.setEditable(false);

        Circle circle = new Circle(250, 80, 30);
        circle.setFill(Color.BLUE);
        circle.addEventHandler(MouseEvent.ANY, this::info);

        VBox root = new VBox();
        root.setAlignment(Pos.TOP_CENTER);
        root.setPadding(new Insets(20, 10, 20, 10));
        root.getChildren().addAll(circle, ta);
        VBox.setMargin(ta, new Insets(20, 0, 0, 0));

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Mouse Events");
        stage.show();
    }

    private void info(MouseEvent e) {
        if (e.getEventType().equals(MouseEvent.MOUSE_MOVED))
            return;
        ta.appendText("****\nButton " +
            e.getButton() + ", event " + e.getEventType() +
            "\nCntrl? "  + e.isControlDown() + "; Shift? " +
            e.isShiftDown() + "\n" + "At x=" + e.getX() +
            " y=" + e.getY() + ", " + e.getClickCount() +
            "-click\n");
    }
}
```

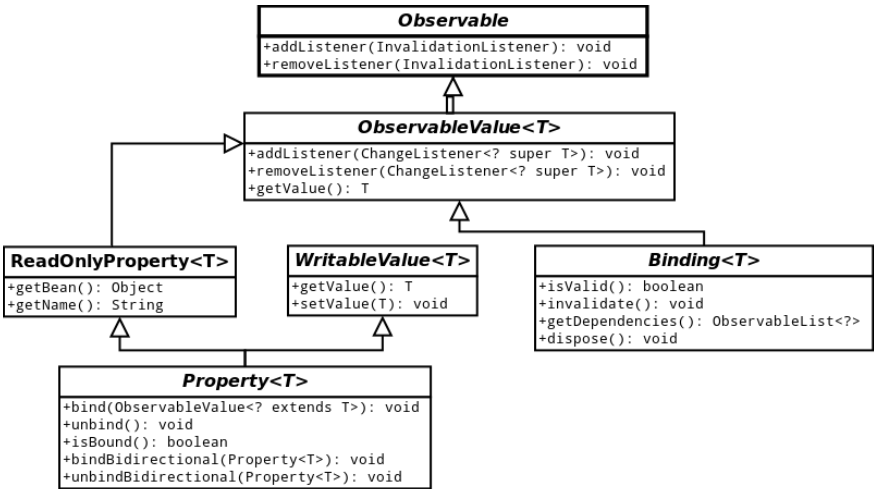
which produces



```
At x=271.0 y=86.0, 1-click
****
Button PRIMARY, event MOUSE_CLICKED
Cntrl? false; Shift? false
At x=271.0 y=86.0, 1-click
****
Button SECONDARY, event MOUSE_PRESSED
Cntrl? false; Shift? false
At x=271.0 y=86.0, 1-click
****
Button SECONDARY, event MOUSE_RELEASED
Cntrl? false; Shift? false
At x=271.0 y=86.0, 1-click
****
Button SECONDARY, event MOUSE_CLICKED
Cntrl? false; Shift? false
At x=271.0 y=86.0, 1-click
****
Button NONE, event MOUSE_EXITED
Cntrl? false; Shift? false
At x=279.0 y=94.0, 0-click
****
Button NONE, event MOUSE_ENTERED
Cntrl? false; Shift? false
At x=244.0 y=108.0, 0-click
****
Button NONE, event MOUSE_EXITED
Cntrl? false; Shift? false
At x=228.0 y=59.0, 0-click
```

12.4 *Properties and bindings*

Properties represent values (properties of a class) wrapped in 'wrapper' classes which provide means for observing their invalidation and modifications. Bindings are also wrapped values (usually *not* properties of any class) but which themselves depend on values of other properties or bindings.


Observable
+addListener(InvalidationListener): void
+removeListener(InvalidationListener): void
ObservableValue<T>
+addListener(ChangeListener<? super T>): void
+removeListener(ChangeListener<? super T>): void
+getValue(): T
ReadOnlyProperty<T>
+getBean(): Object
+getName(): String
WritableValue<T>
+getValue(): T
+setValue(T): void
Binding<T>
+isValid(): boolean
+invalidate(): void
+getDependencies(): ObservableList<?>
+dispose(): void
Property<T>
+bind(ObservableValue<? extends T>): void
+unbind(): void
+isBound(): boolean
+bindBidirectional(Property<T>): void
+unbindBidirectional(Property<T>): void

The root of hierarchy of classes representing properties and bindings is the Observable interface from the *javafx.beans* package. An Observable wraps a value and allows to observe its invalidations (due to modification). It has only two methods: addListener and removeListener which take an object implementing the *functional* InvalidationListener interface with the only abstract method invalidated(Observable). There is no method allowing to get the value held by an Observable; one can only detect its invalidation.

The generic ObservableValue interface from the *javafx.beans.value* extends Observable and adds methods addListener and removeListener which take an object implementing the functional ChangeListener interface with the only abstract method

```
  changed(ObservableValue<? extends T> observable, T oldVal, T newVal)
```

Recall that invalidated from InvalidationListener takes only one argument, so we can call addListener on an ObservableValue with a lambda: the number of arguments indicates whether we are implementing invalidated from InvalidationListener or changed from ChangeListener.

The ObservableValue also adds the getValue method, with obvious functionality. There are other important interfaces extending Observable — they correspond to observable *collections*: ObservableList, ObservableMap, ObservableSet, ObservableArray, with the corresponding listeners (ListChangeListener and similarly for Map, Set and Array).

The interface Observable has many specialized sub-interfaces corresponding to various types of the wrapped value, like ObservableBooleanValue, ObservableDoubleValue, ObservableObjectValue, ObservableStringValue and so on.

The interface ReadOnlyProperty extends ObservableValue and injects two methods: getBean and getName. The first returns the 'owner' of a property, i.e., the object possessing (containing) this property (which can be null if it was not specified explicitly when the object describing the property was created). A property may also be given an immutable name that can later be fetched by getName.

Finally, the Property interface extends ReadOnlyProperty. Additionally, it extends WritableValue which adds setValue methods; Property also adds a few methods on its own, so ultimately a Property wrapping a value of type T has the following methods:

- `addListener(InvalidationListener)`
- `removeListener(InvalidationListener)`
- `addListener(ChangeListener<? super T>)`
- `removeListener(ChangeListener<? super T>)`
- `getValue()`
- `setValue(T)`
- `getBean()`
- `getName()`
- `bind(ObservableValue<? extends T>)`
- `unbind()`
- `isBound()`
- `bindBidirectional(Property<T>)`
- `unbindBidirectional(Property<T>)`

There are many abstract and concrete classes implementing Property interface, like *Type*Property (abstract), Simple*Type*Property, ReadOnly*Type*Wrapper where *Type* can be Double, Integer, Object, String and so on. Instead of setValue and getValue, one can use just set and get — the difference is that (if appropriate) they take/return values of primitive types (int, double, ...), while setValue and getValue operate on values of object types (Integer, Double, String...). Due to boxing/unboxing, we can often use these methods interchangeably.

Let us consider a simple example. We create one IntegerProperty (src) and attach an InvalidationListener to it; the listener prints the message `** Invalidated`, but doesn't try to print the new value. To the constructor of SimpleIntegerProperty, we pass only an initial value, so the property has neither 'owner' nor 'name', as we can see from the last line of the output

Listing 138 JYG-Invalid/Inval.java

```
 1 package invalid;
 2 
 3 import javafx.beans.property.IntegerProperty;
 4 import javafx.beans.property.SimpleIntegerProperty;
 5 
 6 public class Inval {
 7 
 8     public static void main(String[] args) {
 9         IntegerProperty src = new SimpleIntegerProperty(1);
10         src.addListener(observable ->
11                 System.out.println("** Invalidated"));
12 
13         System.out.println("1. " + src);
14         src.set(2);
15         System.out.println("2.");
16         src.set(3);
17         src.set(4);
18         src.set(5);
19         System.out.println("3. getting value: " + src.get());
20         System.out.println("4. " + src);
21         System.out.println("Bean: " + src.getBean() +
22                          ", name: " + src.getName());
23     }
24 }
```

On line 14, we modify the value of our property. As expected, we get a message (form the **InvalidationListener**) that it has been invalidated. Note, however, that after subsequent modifications (lines 16-18), we do *not* get any messages. This illustrates an important feature of **Observable**s — their implementation uses the so called *lazy evaluation*: when the new value is not needed, it is not recalculated and also as long as we do not ask for the value, subsequent invalidations of an already invalidated property are not performed. In line 19, we explicitly ask for the value, so of course now it *will* be recalculated. The output of the program looks therefore like this:

```
1. IntegerProperty [value: 1]
** Invalidated:
2.
3. getting value: 5
4. IntegerProperty [value: 5]
Bean: null, name:
```

Of course, registering a **ChangeListener** to a property precludes the use of lazy evaluation — such listener has to get, as the third argument of its **changed** method, the new value after modification, so it has to be 'eagerly' recalculated.

Objects implementing the **Property** interface are used (although not only) to represent... well, properties of classes, like `width` of geometrical objects, `birth_year` of a **Person** and so on. Classes that expose their properties as **Properties** and conform to some naming conventions are called **JavaFX Beans**. Let us show these conventions on an example. Suppose we want a class to possess a modifiable property height of type double. Then we should

- create a *private* field of type DoubleProperty (which is abstract) and instantiate it with the reference to an object of the concrete type SimpleDoubleProperty; a natural name for such a field would be height (but remember that it's *not* a number — it is a Property wrapping a number!);
- add the method
  `public final double getHeight()`
  which returns `height.get()` as a double;
- add the method
  `public final void setHeight(double)`
  which calls `height.set(double)` (of course, we may choose not to provide this method if we don't want the property to be modifiable);
- add the method
  `public final DoubleProperty heightProperty()`
  which returns height (so a Property as such, not its wrapped value!);
- add a default constructor — this is not mandatory, but many Java frameworks expect its existence.

As usually, one should also consider overriding the **equals** and **hashCode** methods.

These requirements are different for properties that should not be directly modifiable by the client code, although they still can be changed internally, by methods of the class. Suppose a property `name` of type String is such a property. Then we should proceed as follows:

- create a *private* field of the (concrete) type ReadOnlyStringWrapper and instantiate it — a natural name for such a field would be name. Objects of such types (there are many of them, with Integer, Double, Object etc. instead of String) have 'normal' getters and setters;
- add the method
  `public final String getName()`
  which returns `name.get()` as a String;
- add the method
  `public final ReadOnlyStringProperty nameProperty()`
  which returns name.getReadOnlyProperty(). The client code will get an object that can be used for binding to, reading the value, attaching listeners, but will not allow modifications of the wrapped value. However, modifications are still possible for methods of the class by invoking setters on the (private) wrapper field (name in our case). Such modifications will be visible by the client code, because values held by the wrapper object and by the read-only property returned by **nameProperty** method are synchronized;
- add a default constructor — this is not mandatory, but many Java frameworks expect it.

As an example, consider the following program, which defines a JavaFX-bean class Person:

Listing 139 JXF-Props/FXProps.java

```java
package jfxprops;

```

```java
import java.util.Objects; // for hash function
import javafx.beans.property.ObjectProperty;
import javafx.beans.property.ReadOnlyIntegerProperty;
import javafx.beans.property.ReadOnlyIntegerWrapper;
import javafx.beans.property.SimpleObjectProperty;
import javafx.beans.property.StringProperty;
import javafx.beans.property.SimpleStringProperty;
import javafx.scene.paint.Color;

public class FXProps {
    public static void main(String[] args) {
        Person mary = new Person("Mary", 17, Color.PINK);
        ReadOnlyIntegerProperty p = mary.ageProperty();
        ((Person)p.getBean()).updateAge(18);
        System.out.println(mary);
        mary.setFavColor(Color.PURPLE);
        System.out.println(mary);
    }
}

class Person { // an FX bean
    private final StringProperty name;
    private final ObjectProperty<Color> favColor;
    private final ReadOnlyIntegerWrapper age;

    public Person(String n, int a, Color c) {
        name = new SimpleStringProperty(this, "name", n);
        favColor = new SimpleObjectProperty<>(this,
                                              "favColor", c);
        age  = new ReadOnlyIntegerWrapper(this, "age", a);
    }
      // default constructor not mandatory but should exist
    public Person() {
        this("", 0, Color.WHITE);
    }

      // RW properties
    public final String getName() {
        return name.get();
    }
    public final StringProperty nameProperty() {
        return name;
    }
    public final void setName(String n) {
        name.set(n);
    }

    public final Color getFavColor() {
        return favColor.get();
    }
    public final ObjectProperty<Color> favColorProperty() {
        return favColor;
    }
    public final void setFavColor(Color c) {
        favColor.set(c);
    }
      // RO property (no public setter)
    public final int getAge() {
        return age.get();
    }
    public final ReadOnlyIntegerProperty ageProperty() {
        return age.getReadOnlyProperty(); // not just age!
    }

      // it's still possible to modify a RO
      // property, but only within the class
    public final void updateAge(int a) {
        age.set(a);
    }

    @Override
    public String toString() {
        return name.get() + "(" + age.get() +
                ") - " + favColor.get();
    }
    @Override
    public boolean equals(Object other) {
        if (other == null || getClass() != other.getClass())
            return false;
        if (this == other) return true;
        Person p = (Person)other;
        return name.get().equals(p.getName())         &&
               age.get() == p.getAge()                &&
               favColor.get().equals(p.getFavColor());
    }
    @Override
    public int hashCode() {
        return Objects.hash(
                name.get(), favColor.get(), age.get());
    }
}
```

We have three properties here: name, favColor and age. The first two are modifiable, the third is not. Note that creating property objects, we pass to the constructor not only initial values, but also, as the first two arguments, their ‘owner’ (which is **this**) and name. This is not required, but often useful because having the property, we can access its ‘owner’ by calling the **getBean** method on it. Note also that **ObjectProperty** is generic — we thus specify the type parameter in angle brackets. The age property is read-only, but methods of the class (**updateAge** in our case) can modify it by referring directly to the private ‘wrapper’ field age.

The program prints

```
Mary(18) - 0xffc0cbff
Mary(18) - 0x800080ff
```

(hex numbers printed here are rgbα components of Mary’s current favorite color).

Properties may be bound to other **ObservableValue** objects. Calling on a property the method **bind** with an argument of type **ObservableValue** (or **Property** which inherits from it) creates a **binding**, in other words a dependency between the property object and the observable value. From now on, we cannot directly modify the property — calling **set** or **setValue** on it will cause a **RuntimeException** to be thrown. If the observable value is modified, the property becomes invalidated, however calling the **get** or **getValue** methods on the property will always return the current value of the **ObservableValue** object (this is again an example of *lazy evaluation*). As holding such bindings may be quite costly, we should break the dependency as soon as it becomes unnecessary (by calling the **unbind** method on the bound property). One can always check if a property is or is nor bound by invoking the **isBound** method.

Two property objects may also be mutually bound to each other (**bidirectional binding**). If p and q are two properties, then after p.bindBidirectional(q) the two properties are mutually bound — modification of one of them modifies the other accordingly.

The following example illustrates both uni- and bidirectional bindings:

Listing 140 JYH-BindProps/BindProps.java

```java
package bindprops;

import javafx.beans.property.IntegerProperty;
import javafx.beans.property.SimpleIntegerProperty;

public class BindProps {
    public static void main(String[] args) {
        IntegerProperty p = new SimpleIntegerProperty(1);
        IntegerProperty q = new SimpleIntegerProperty();
        q.bind(p);
        System.out.println("1. " + q.get());
        p.set(2);
        System.out.println("2. " + q.get());
        q.unbind();

        p.bindBidirectional(q);
        p.set(3);
        System.out.println("3. p=" + p.get() +
                           ", q=" + q.get());
        q.set(4);
        System.out.println("4. p=" + p.get() +
                           ", q=" + q.get());
        p.unbindBidirectional(q);
        p.set(5);
        q.set(6);
        System.out.println("5. p=" + p.get() +
                            ", q=" + q.get());
    }
}
```

The program prints

```
1. 1
2. 2
3. p=3, q=3
4. p=4, q=4
5. p=5, q=6
```

Binding properties to observable values is extremely useful, but still there is a serious downside: a property may be bound to only one other value and then their values are synchronized, but basically the same. Very often, we would like to have a value which depends on more than one other observable values and whose value is some, arbitrary complex, function of these other values. That's what **Binding**s are for. Objects implementing the **Binding** interface also, like **Properties**, represent a value equipped with some additional functionality. As we have seen in the figure depicting the class hierarchy (p. 284), a **Binding** is an **ObservableValue**, but is neither **WritableValue** nor **ReadOnlyProperty**. This means that we cannot modify it directly by calling **set** or **setValue** (we *can* invalidate it, though). Also, it doesn't have any 'owner', i.e., it doesn't represent a property of a class (there is no **getBean** method). On the other hand, it can depend, in a non-trivial way, on many **ObservableValues**, **Properties**, other **Binding**s, and even regular Java variables.

Object of the **Binding** type can be created in three ways. Let us start with perhaps the most flexible one.

The **Binding** interface is implemented by several *abstract* classes depending on the type of the wrapped value: **BooleanBinding**, **FloatBinding**, **DoubleBinding**, **IntegerBinding**, **LongBinding**, **StringBinding**, **ObjectBinding**, **ListBinding**, **MapBinding**, **SetBinding**. Numerical types (**DoubleBinding**, **IntegerBinding**...) are all specialized through the **Number** supertype, so we can bind to, say, **DoubleBinding**, an **IntegerProperty**, and so on.

In order to create a binding of one of these (abstract) types, we have to extend it providing an implementation of their sole abstract method **computeValue**; it doesn't take any arguments, but of course we have to make values that the result is to depend on visible inside its body. **Important:** Implementing our concrete class, we have to call, in the first line of the constructor,

```java
super.bind(v1, v2, v3);
```

where `v1`, `v2` and so on are *dependencies*, i.e., **Observable** objects the value of the binding depends on (one or more). If, what is quite common, we rather use an object of an anonymous class, so we cannot define any constructor, we can replace it by defining a (non-static) initializer block, as illustrated in the example below:

Listing 141 JYI-Bindings/RectArea.java

```java
package rectarea;

import javafx.beans.binding.DoubleBinding;
import javafx.beans.property.DoubleProperty;
import javafx.beans.property.SimpleDoubleProperty;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;

public class RectArea {
    public static void main(String[] args) {
        DoubleProperty x = new SimpleDoubleProperty(3.0);
        DoubleProperty y = new SimpleDoubleProperty(4.0);
        DoubleBinding area = new DoubleBinding() {
            {
                super.bind(x, y);
            }
            @Override
            protected double computeValue() {
                return x.get() * y.get();
            }

              // getDependencies and dispose - if needed.
              // They have empty default implementation.
              // Just for illustration...
            @Override
            public ObservableList getDependencies() {
                return FXCollections .observableArrayList(
                                                     x, y);
            }

            @Override
            public void dispose() {
                super.unbind(x, y);
            }
        };

        System.out.println(area.get());
        x.set(5);
        y.set(6);
        System.out.println(area.get());

          // just for illustration...
        ObservableList<?> list = area.getDependencies();
        for (Object ob : area.getDependencies())
            System.out.println(
                    ((SimpleDoubleProperty)ob).get());
        area.dispose();
    }
}
```

which prints

```
12.0
```

```
30.0
5.0
6.0
```

Abstract classes mentioned above have, among many others, non-abstract (already implemented) methods **dispose**, which is supposed to release any resources used by the binding, and **getDependencies** which should return all dependencies (those listed in the invocation of `super.bind(...)`) as an unmodifiable **ObservableList**. However, the implementation of these methods, although exists, is empty, so if they are desired, they have to be overridden, as shown in the example program above. Overriding the **getDependencies** method is not recommended, except for debugging, monitoring the behavior of a program etc. during development; also **dispose** is seldom used.

The second way of creating a binding is by using one of the *static* factory methods of the **Bindings** (note the 's') utility class. There are more than 230 such methods, but they all follow a similar pattern, so it isn't hard to guess their names and types of parameters. For example, for numerical types, we have, among others, static methods (called on **Bindings**) which create bindings depending on two values and representing their sum. They all are `static public`; below we show their return type, name, and parameters:

```
NumberBinding add(ObservableNumberValue n1,
                  ObservableNumberValue n2)
DoubleBinding add(ObservableNumberValue n, double d)
DoubleBinding add(double d, ObservableNumberValue n)
NumberBinding add(ObservableNumberValue n, float f)
NumberBinding add(float f, ObservableNumberValue n)
NumberBinding add(ObservableNumberValue n, long l)
NumberBinding add(long l, ObservableNumberValue n)
NumberBinding add(ObservableNumberValue n, int i)
NumberBinding add(int i, ObservableNumberValue n)
```

As one can see, at least one of the parameters is always of the type **ObservableNumberValue** which is an interface implemented by classes **DoubleBinding**, **DoubleProperty**, **DoubleExpression** and so on for other numerical types. The other parameter, the first or the second, can be just a number (**int**, **float**, **double** or **long**). Similar sets of static methods correspond to subtraction (**subtract**), multiplication (**multiply**) and division (**divide**). For example. the following program

```
import javafx.beans.property.DoubleProperty;
import javafx.beans.property.SimpleDoubleProperty;
import javafx.beans.binding.DoubleBinding;
import javafx.beans.binding.Bindings;

public class Bind {
    public static void main(String[] args) {
        DoubleProperty a = new SimpleDoubleProperty(3),
                       b = new SimpleDoubleProperty(4);
        DoubleBinding average =
                  Bindings.divide(Bindings.add(a, b), 2.0);
        System.out.println(average.get());
        a.set( 5);
```

```
            b.set(12);
            System.out.println(average.get());
        }
    }
```

compiles and prints

```
    3.5
    8.5
```

If we call many static methods from the **Bindings** class, it would be convenient to import them statically at the beginning

```
    import static javafx.beans.binding.Bindings.*;
```

For numerical arguments, we also have several factory methods returning a **BooleanBinding**, like **equal**, **greaterThan**, **greaterThanOrEqual**, etc. Similarly, for boolean arguments, we can use **and**, **or** and **not**, as in the following example, where isBinAC represents **true** if, and only if, $b \in [a, c]$:

```
    import javafx.beans.property.DoubleProperty;
    import javafx.beans.property.SimpleDoubleProperty;
    import javafx.beans.binding.BooleanBinding;
    import javafx.beans.binding.NumberBinding;
    import javafx.beans.binding.Bindings;

    public class Bind {
        public static void main(String[] args) {
            DoubleProperty a = new SimpleDoubleProperty(3),
                           b = new SimpleDoubleProperty(4),
                           c = new SimpleDoubleProperty(5);
            BooleanBinding isBinAC =
                  Bindings.and(Bindings.greaterThanOrEqual(b, a),
                               Bindings.lessThanOrEqual(b, c));
            System.out.println(isBinAC.get());
            b.set(6);
            System.out.println(isBinAC.get());
            NumberBinding mn = Bindings.min(Bindings.min(a,b),c);
            NumberBinding mx = Bindings.max(Bindings.max(a,b),c);
            System.out.println("mn=" + mn.getValue() +
                                 ", mx=" + mx.getValue());
        }
    }
```

which prints

```
    true
    false
    mn=3.0, mx=6.0
```

and demonstrates also factory methods **min** and **max**.

For **ObservableStringValues** one can use **equalIgnoreCase** and **notEqualIgnoreCase**, for any **ObservableObjectValues** — **isNull** and **isNotNull**, etc.

Another useful set of factory methods, **create*Type*Binding**, allow us to create bindings with custom implementation of the function computing the value based on the values of dependencies without extending any class: we just pass the dependencies and, as the first argument, an implementation of the **Callable** interface calculating the result of an appropriate type. The interface has one abstract method, **call**, and is therefore a functional interface which can be conveniently implemented by a lambda, as below (compare this program with Listing 141):

```
import javafx.beans.binding.DoubleBinding;
import javafx.beans.property.DoubleProperty;
import javafx.beans.property.SimpleDoubleProperty;
import javafx.beans.binding.Bindings;

public class Bind {
    public static void main(String[] args) {
        DoubleProperty x = new SimpleDoubleProperty(3.0);
        DoubleProperty y = new SimpleDoubleProperty(4.0);
        DoubleBinding area = Bindings.createDoubleBinding(
                              () -> x.get()*y.get(), x, y);
        System.out.println(area.get());
        x.set(5);
        y.set(6);
        System.out.println(area.get());
    }
}
```

There are many more factory methods of the **Bindings** class worth checking in the documentation (e.g., **when** or a family of the **select** methods).

Finally, bindings may be created by using the so called **fluent API**. Let us demonstrate it on an example of type **Double**, but of course it can be used for other types as well. There is a common abstract super-class of **DoubleBinding** and **ReadOnlyDoubleProperty** called **DoubleExpression** (note that **ReadOnlyDoubleProperty** is in turn a base class of **DoubleProperty**, so properties *are* expressions). This class defines several *non-static* methods with names similar to those from the **Bindings** class, but this time, as they are non-static, without one of the parameters (whose rôle is played by the 'hidden' **this** argument). The methods return bindings (therefore, expressions) so they can be applied in chains of invocations. For example, for two properties prop1 and prop2

```
prop1.add(prop2).divide(2.0)
```

is (almost) equivalent to

```
Bindings.divide(Bindings.add(prop1, prop2), 2.0)
```

(return types in fluent API are sometimes more specific than for the corresponding static methods of the **Bindings** class).

Of course, one can mix static factory methods of the **Bindings** class with the fluent API approach, like in the following program

Listing 142 JXL-Binding/BindMix.java

```
package bindings;

import javafx.beans.property.DoubleProperty;
```

```java
import javafx.beans.property.SimpleDoubleProperty;
import javafx.beans.binding.NumberBinding;
import javafx.beans.binding.Bindings;

public class BindMix {
    public static void main(String[] args) {
        DoubleProperty a = new SimpleDoubleProperty(3);
        DoubleProperty b = new SimpleDoubleProperty(4);
        DoubleProperty c = new SimpleDoubleProperty(5);
        NumberBinding average =
                  Bindings.divide(a.add(b).add(c),3.0);
        System.out.println(average.getValue());
        a.setValue( 5);
        b.setValue(12);
        c.setValue(13);
        System.out.println(average.getValue());
    }
}
```

which prints

```
4.0
10.0
```

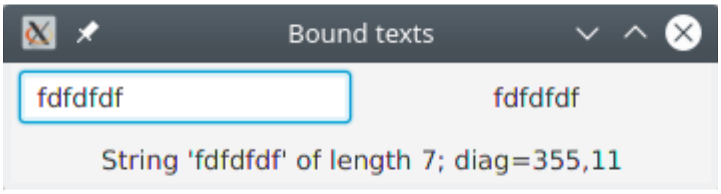
Let us consider another example: we have here two labels (variables lt and cc and a text field (tt). Both labels and text fields have property text: on line 40, we bind text property of the label with the text property of the text field. Therefore, every time we modify the text field, the text on the label is also modified accordingly. On the next line we bind the text property of the second label with a custom binding of type **StringBinding**. The value of the binding is a string with information on the content of the string in the tt text field, its length and also the diagonal of the root FX node, which is an object of type **GridPane** (with properties width and height):

Listing 143 JXN-BindText/BindText.java

```java
1  package bindtext;
2  
3  import javafx.application.Application;
4  import javafx.beans.binding.StringBinding;
5  import javafx.geometry.HPos;
6  import javafx.geometry.Insets;
7  import javafx.scene.Scene;
8  import javafx.scene.control.Label;
9  import javafx.scene.control.TextField;
10 import javafx.scene.layout.ColumnConstraints;
11 import javafx.scene.layout.GridPane;
12 import javafx.stage.Stage;
13 
14 public class BindText extends Application {
15 
```

```
    @Override
    public void start(Stage stage) {
        TextField tt = new TextField("Enter something...");
        Label     lt = new Label();
        Label     cc = new Label();

        GridPane root = new GridPane();
        root.add(tt,  0, 0, 1, 1);
        root.add(lt,  1, 0, 1, 1);
        root.add(cc,  0, 1, 2, 1);

        root.setPadding(new Insets(4));
        root.setHgap(10);
        root.setVgap(10);
        GridPane.setHalignment(lt, HPos.CENTER);
        GridPane.setHalignment(cc, HPos.CENTER);

        int[] percWidth = {50, 50};
        for (int col = 0; col < 2; ++col) {
            ColumnConstraints c = new ColumnConstraints();
            c.setPercentWidth(percWidth[col]);
            root.getColumnConstraints().add(c);
        }

        lt.textProperty().bind(tt.textProperty());
          // now custom binding
        cc.textProperty().bind(new StringBinding() {
            { // pseudo constructor!
                super.bind(tt.textProperty(),
                           tt.lengthProperty(),
                           root.widthProperty(),
                           root.heightProperty());
            }
            @Override
            protected String computeValue() {
                double d = Math.sqrt(
                    root.getHeight()*root.getHeight() +
                    root.getWidth()*root.getWidth());
                return "String '" + tt.getText() +
                "' of length " + tt.getLength() +
                "; diag=" + String.format("%.2f", d);
            }
        });

        Scene scene = new Scene(root);
        stage.setTitle("Bound texts");
        stage.setScene(scene);
        stage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

The program displays



fdfdfdf

fdfdfdf

String 'fdfdfdf' of length 7; diag=355,11

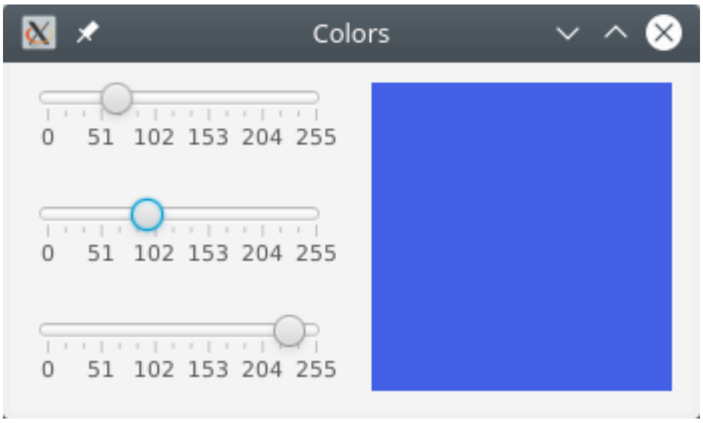
The next example illustrates ObjectBinding (specialized for Color). On line 31, we create a binding depending on value properties of three Sliders: red, green and blue. The first argument of the createObjectBinding function is an implementation of Callable interface (here passed as a lambda) returning the result value — here it is the color described by the three value properties of the sliders. We then bind fill property of a rectangle (the color of its interior) to the binding just created. In this way, changes of the position of any of the sliders are immediately reflected as the color of our rectangle:

Listing 144 JXP-BindColor/BindColor.java

```
 1 package bindcolor;
 2 
 3 import javafx.application.Application;
 4 import javafx.beans.binding.ObjectBinding;
 5 import javafx.beans.binding.Bindings;
 6 import javafx.geometry.Insets;
 7 import javafx.scene.Scene;
 8 import javafx.scene.control.Slider;
 9 import javafx.scene.layout.GridPane;
10 import javafx.scene.layout.StackPane;
11 import javafx.scene.paint.Color;
12 import javafx.scene.shape.Rectangle;
13 import javafx.stage.Stage;
14 
15 public class BindColor extends Application {
16     public static void main(String[] args) {
17         launch(args);
18     }
19 
20     @Override
21     public void start(Stage stage) {
22         Slider red   = getSlider(66);
23         Slider green = getSlider( 0);
24         Slider blue  = getSlider( 0);
25         StackPane pane = new StackPane();
26         pane.setMinWidth(150);
```

```
        Rectangle rec = new Rectangle(120, 120);
        pane.getChildren().add(rec);


        ObjectBinding<Color> colBind =
            Bindings.createObjectBinding(
                () -> Color.rgb(
                          (int)red.valueProperty().get(),
                          (int)green.valueProperty().get(),
                          (int)blue.valueProperty().get()
                      ),
                red.valueProperty(),
                green.valueProperty(),
                blue.valueProperty());
        rec.fillProperty().bind(colBind);
        GridPane root = new GridPane();
        root.add(red,   0, 0, 1, 1);
        root.add(green, 0, 1, 1, 1);
        root.add(blue,  0, 2, 1, 1);
        root.add(pane,  1, 0, 1, 3);
        root.setPadding(new Insets(10));
        root.setHgap(20);
        root.setVgap(20);

        stage.setTitle("Bound color");
        stage.setScene(new Scene(root));
        stage.show();
    }

    private Slider getSlider(int iniVal) {
        Slider slider = new Slider(0, 255, iniVal);
        slider.setShowTickMarks(true);
        slider.setShowTickLabels(true);
        slider.setMajorTickUnit(51);
        slider.setMinorTickCount(2);
        slider.setBlockIncrement(1);
        slider.setMinWidth(150);
        return slider;
    }
}
```

The program displays



12.5 Effects

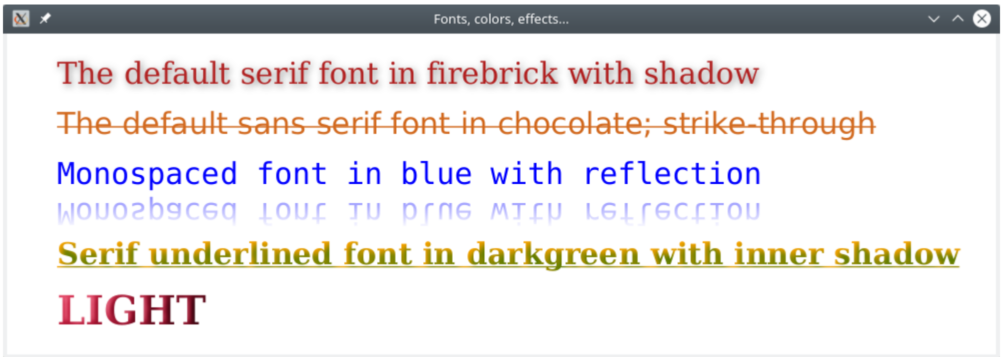
Effects can be applied to nodes to modify their appearance. Various kind of effects are represented by several classes form the ***javafx.scene.effect*** package: these are, among others **Blend**, **Bloom**, **ColorAdjust**, **DropShadow**, **Glow**, **InnerShadow**, **Lighting**, **Reflection**, **Shadow**, **SepiaTone**...

Let's consider just one an example illustrating simple text effects:

Listing 145 JXE-Fonts/FontsETC.java

```java
package fontsetc;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.effect.DropShadow;
import javafx.scene.effect.InnerShadow;
import javafx.scene.effect.Light;
import javafx.scene.effect.Lighting;
import javafx.scene.effect.Reflection;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.text.Font;
import javafx.scene.text.FontWeight;
import javafx.scene.text.Text;
import javafx.stage.Stage;

public class FontsETC extends Application {
    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("Fonts, colors, effects...");

        System.out.println("**** Font families:");
        Font.getFamilies().stream()
                          .forEach(System.out::println);
        System.out.println("**** Fonts: ");
        Font.getFontNames().stream()
                           .forEach(System.out::println);

        Text t1 = new Text(50, 50, "The default serif " +
                         "font in firebrick with shadow");
        t1.setFont(Font.font("Serif", 30));
        t1.setFill(Color.FIREBRICK);
        DropShadow dropShadow = new DropShadow();
        dropShadow.setOffsetX(2.0f);
        dropShadow.setOffsetY(2.0f);
        dropShadow.setColor(Color.rgb(50, 50, 50, 0.6));
        t1.setEffect(dropShadow);

        Text t2 = new Text(50, 100, "The default sans " +
               "serif font in chocolate; strike-through");
        t2.setFont(Font.font("SanSerif", 30));
        t2.setFill(Color.CHOCOLATE);
        t2.setStrikethrough(true);

        Text t3 = new Text(50, 150, "Monospaced font in " +
                                    "blue with reflection");
        t3.setFont(Font.font("Monospaced", 30));
        t3.setFill(Color.BLUE);
        Reflection ref = new Reflection();
        ref.setFraction(0.9f);
        ref.setTopOffset(3);
        t3.setEffect(ref);

        Text t4 = new Text(50, 230, "Serif underlined fo" +
                       "nt in darkgreen with inner shadow");
        t4.setFont(Font.font("Serif", FontWeight.BOLD, 30));
        t4.setFill(Color.DARKGREEN);
        t4.setUnderline(true);
        InnerShadow inShadow = new InnerShadow();
        inShadow.setOffsetX(4);
        inShadow.setOffsetY(4);
        inShadow.setColor(Color.ORANGE);
        t4.setEffect(inShadow);

        Text t5 = new Text(50, 290, "LIGHT");
        t5.setFont(Font.font("Serif", FontWeight.BOLD, 40));
        t5.setFill(Color.CRIMSON);
        Light.Point light = new Light.Point(
                                0, 0, 80, Color.WHITE);
        Lighting lighting2 = new Lighting(light);
        t5.setEffect(lighting2);

        Pane root = new Pane(t1, t2, t3, t4, t5);
        Scene scene = new Scene(root, 900, 320, Color.WHITE);

        primaryStage.setScene(scene);
        primaryStage.show();
    }

    public static void main(String args[]) {
        Application.launch(args);
    }
}
```

which produces



The default serif font in firebrick with shadow

~~The default sans serif font in chocolate; strike-through~~

Monospaced font in blue with reflection

Serif underlined font in darkgreen with inner shadow

LIGHT

12.6 *Lists, tables and trees*

Class ListView describes, as its name suggests, a view of a list of objects. First, the list to be shown must be made an **observable list**, i.e., a list that allows attaching listeners which track its changes and fire appropriate events when they occur. Such lists implement the interface ObservableList which is very similar to 'normal' List. In particular, you can use methods like **add**, **addAll**, **remove** etc., but also, as it is *observable*, **addListener** and **removeListener**. Objects of classes implementing this interface can be easily created in several ways, the most common way being a call of a generic static function of class FXCollections, like this (assuming mary, john,... are references to Persons)

```
    ObservableList<Person> list =
        FXCollections.observableArrayList(
            mary, john, kate, new Person("Cindy")
        );
```

or, assuming that coll is a 'normal' collection of references to Persons

```
    ObservableList<Person> list =
        FXCollections.observableArrayList(coll);
```

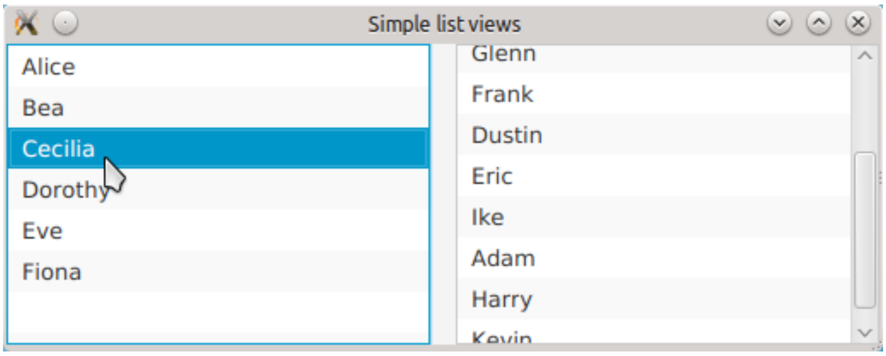
Having created an observable list, it is enough to pass it to the constructor of class ListView; the constructor will automatically register the view as a listener of the observable list so it will react to all modifications of the list 'all by itself'. It is also possible to create a ListView first, and then call the method **setItems** passing an observable list.

Let's see a very simple example. We show two list views side-by-side. The example demonstrates how we can attach a listener reacting to selecting an item of a list:

Listing 146 JYC-SimpleListView/SimpleListView.java

```java
package simplelistview;

import java.util.Set;
import javafx.application.Application;
import javafx.beans.value.ChangeListener;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.scene.Scene;
import javafx.scene.control.ListView;
import javafx.scene.layout.TilePane;
import javafx.stage.Stage;

public class SimpleListView extends Application {
    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage stage) {
        ObservableList<String> list1 =
            FXCollections.observableArrayList(
                "Alice", "Bea", "Cecilia", "Dorothy", "Eve"
            );
        list1.add("Fiona");

        Set<String> set =
            Set.of(
                "Adam", "Bruce", "Chuck", "Dustin", "Eric",
                "Frank", "Glenn", "Harry", "Ike", "Joshua"
            );
        ObservableList<String> list2 =
            FXCollections.observableArrayList(set);
        list2.add("Kevin");

        ListView<String> view1 = new ListView<>(list1);
        ListView<String> view2 = new ListView<>();
        view2.setItems(list2);

        view1.setPrefHeight(130);
        view2.setPrefHeight(200);

        ChangeListener<String> listener =
            (obsVal, oldV, newV) -> {
                System.out.println("Selected: " + newV);
            };

        view1.getSelectionModel().selectedItemProperty()
             .addListener(listener);
        view2.getSelectionModel().selectedItemProperty()
             .addListener(listener);

        TilePane root = new TilePane(view1, view2);
        root.setHgap(15);
        root.setPrefColumns(2);
        stage.setScene(new Scene(root));
        stage.setTitle("Simple list views");
        stage.show();
    }
}
```

The program displays


Simple list views

Alice
Bea
Cecilia
Dorothy
Eve
Fiona

Glenn
Frank
Dustin
Eric
Ike
Adam
Harry
Kevin

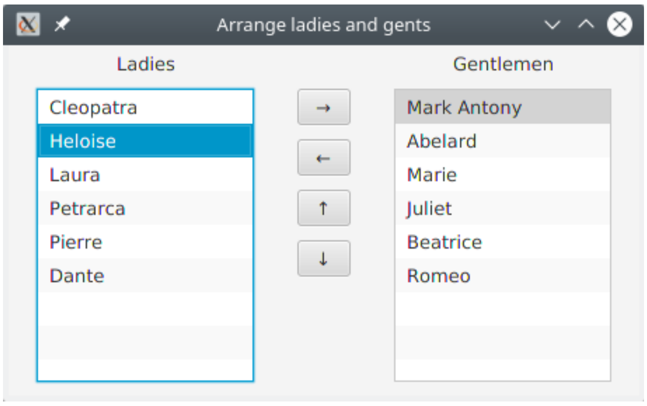
Next example is a little more complex: we also display two lists, but also a pane with four buttons. Clicking a button, the user can transfer an item from one list to the other or move the selected item up or down:

Listing 147 JXM-ListView/ListEx.java

```
package lists;

import javafx.application.Application;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.geometry.HPos;
import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.ListView;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.ColumnConstraints;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.Priority;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;
```

```

public class ListEx extends Application {

    public static void main(String[] args) {
        launch(args);
    }

    ListView<String> lView = null;
    ListView<String> gView = null;
    Button lrBut = new Button(" \u2192 "); // ->
    Button rlBut = new Button(" \u2190 "); // <-
    Button upBut = new Button(" \u2191 "); // up
    Button dnBut = new Button(" \u2193 "); // down

    ListView<String> lastView = null;

    public void init() {
        ObservableList<String> ladies =
                FXCollections.observableArrayList(
                    "Cleopatra", "Laura", "Petrarca",
                    "Romeo", "Pierre", "Dante",  "Heloise");
        ObservableList<String> gents =
                FXCollections.observableArrayList(
                    "Abelard", "Marie", "Juliet",
                    "Mark Antony", "Beatrice");
        lView = new ListView<>(ladies);
        gView = new ListView<>(gents);

        EventHandler<ActionEvent> hHandler =
            new EventHandler<ActionEvent>() {
                @Override
                public void handle(ActionEvent event) {
                    ListView<String> fromView = null;
                    ObservableList<String> fromList = null,
                                             toList = null;
                    if (event.getSource().equals(rlBut)) {
                        fromView = gView;
                        fromList = gents;
                        toList   = ladies;
                    } else if (event.getSource()
                                     .equals(lrBut)) {
                        fromView = lView;
                        fromList = ladies;
                        toList   = gents;
                    } else return;
                    String s = fromView.getSelectionModel()
                                       .getSelectedItem();
                    if (s != null) {
                        fromView.getSelectionModel()
                                .clearSelection();
                        fromList.remove(s);
                        toList.add(s);
                    }
                }
            };
        rlBut.setOnAction(hHandler);
        lrBut.setOnAction(hHandler);

        EventHandler<ActionEvent> vHandler =
            new EventHandler<ActionEvent>() {
                @Override
                public void handle(ActionEvent event) {
                    String s = lastView.getSelectionModel()
                                       .getSelectedItem();
                    if (s == null) return;
                    boolean up = event.getSource()
                                       .equals(upBut);
                    int ind = lastView.getSelectionModel()
                                       .getSelectedIndex();
                    if (ind < 0) return;
                    int nextInd = up ? ind-1 : ind + 1;
                    int last = lastView.getItems().size()-1;
                    if (up && ind == 0) {
                        nextInd = 0;
                    } else if (!up && ind == last) {
                        nextInd = last;
                    } else {
                        ObservableList<String> list =
                                        lastView.getItems();
                        list.set(ind, list.get(nextInd));
                        list.set(nextInd, s);
                    }
                    lastView.requestFocus();
                    lastView.getSelectionModel()
                            .clearAndSelect(nextInd);
                }
            };
        upBut.setOnAction(vHandler);
        dnBut.setOnAction(vHandler);
    }

    @Override
    public void start(Stage stage) {
        stage.setTitle("Arrange ladies and gents");
        BorderPane root = new BorderPane();
        Scene scene = new Scene(root, 450, 250);

        GridPane grid = new GridPane();
        grid.setPadding(new Insets(5, 20, 10, 20));
        grid.setHgap(10);
        grid.setVgap(10);
        ColumnConstraints col1 =
            new ColumnConstraints(150,150,Double.MAX_VALUE);
        ColumnConstraints col2 =
            new ColumnConstraints(80);
        ColumnConstraints col3 =
            new ColumnConstraints(150,150,Double.MAX_VALUE);
        col1.setHgrow(Priority.ALWAYS);
        col3.setHgrow(Priority.ALWAYS);
        grid.getColumnConstraints().addAll(col1, col2, col3);

        Label lLab = new Label("Ladies");
        GridPane.setHalignment(lLab, HPos.CENTER);
        grid.add(lLab, 0, 0);
        lView.focusedProperty().addListener((o, p, n) -> {
            if (n) lastView = lView;
        });

        Label gLab = new Label("Gentlemen");
        GridPane.setHalignment(gLab, HPos.CENTER);
        grid.add(gLab, 2, 0);
        gView.focusedProperty().addListener((o, p, n) -> {
            if (n) lastView = gView;
        });

        grid.add(lView, 0, 1);
        grid.add(gView, 2, 1);

        VBox vbox = new VBox();
        vbox.setSpacing(10);
        vbox.setAlignment(Pos.TOP_CENTER);
        vbox.getChildren().addAll(lrBut,rlBut,upBut,dnBut);
        grid.add(vbox, 1, 1);

        root.setCenter(grid);
        GridPane.setVgrow(root, Priority.ALWAYS);
        stage.setScene(scene);
        stage.show();
    }
}
```

The GUI displayed by the program looks like this:



Now, let's have a look at another example of a list view — this time we use a 'cell factory' to render the cells of the list view in a special way (see lines 67-73). We set a cell factory for our list view by implementing the functional interface Callback: its call method has to return an object of type ListCell which will be created by the platform to render a cell of the list view when needed. This class inherits (indirectly) from Labeled and may be treated more or less like a label; in particular we can set a text and 'graphic' (as shown on line 91) although, contrary to what the name suggests, it doesn't have to be a graphic — in fact, it can be an object of any class inheriting from Node. In the example, we return an object of the custom class ColorBox: it inherits from ListCell and has to override the method updateItem. When overriding this method, we have to

- call super.updateItem;
- check if the first argument is not null and the second (empty) is not true: if so, we set both text and graphic to null (empty equal to true means that the cell to be rendered belongs to an empty row and should not be visible at all). Otherwise, we just set text and/or 'graphic', as we would do for a label.

In the example, we also set a tooltip for objects of our class — this is, of course, not necessary but might be helpful for the users (here the tooltips just display characteristics of colors).

Listing 148 JXZ-ColsList/CssColors.java

```
1  package csscolors;
2  
3  import javafx.application.Application;
4  import javafx.collections.FXCollections;
5  import javafx.collections.ObservableList;
6  import javafx.scene.Scene;
7  import javafx.scene.control.ListCell;
8  import javafx.scene.control.ListView;
9  import javafx.scene.control.Tooltip;
10 import javafx.scene.paint.Color;
11 import javafx.scene.shape.Rectangle;
12 import javafx.stage.Stage;
13 import javafx.util.Callback;
14 
15 public class CssColors extends Application {
16     final static ObservableList<String> data =
        FXCollections.observableArrayList(
            "aliceblue", "antiquewhite", "aqua",
            "aquamarine", "azure", "beige", "bisque",
            "black", "blanchedalmond", "blue", "blueviolet",
            "brown", "burlywood", "cadetblue", "chartreuse",
            "chocolate", "coral", "cornflowerblue",
            "cornsilk", "crimson", "cyan", "darkblue",
            "darkcyan", "darkgoldenrod", "darkgray",
            "darkgreen", "darkgrey", "darkkhaki",
            "darkmagenta", "darkolivegreen", "darkorange",
            "darkorchid", "darkred", "darksalmon",
            "darkseagreen", "darkslateblue", "darkslategray",
            "darkslategrey", "darkturquoise", "darkviolet",
            "deeppink", "deepskyblue", "dimgray", "dimgrey",
            "dodgerblue", "firebrick", "floralwhite",
            "forestgreen", "fuchsia", "gainsboro",
            "ghostwhite", "gold", "goldenrod", "gray",
            "green", "greenyellow", "grey", "honeydew",
            "hotpink", "indianred", "indigo", "ivory",
            "khaki", "lavender", "lavenderblush",
            "lawngreen", "lemonchiffon", "lightblue",
            "lightcoral", "lightcyan",
            "lightgoldenrodyellow", "lightgray",
            "lightgreen", "lightgrey", "lightpink",
            "lightsalmon", "lightseagreen", "lightskyblue",
            "lightslategray", "lightslategrey",
            "lightsteelblue", "lightyellow", "lime",
            "limegreen", "linen", "magenta", "maroon",
            "mediumaquamarine", "mediumblue", "mediumorchid",
            "mediumpurple", "mediumseagreen",
            "mediumslateblue", "mediumspringgreen",
            "mediumturquoise", "mediumvioletred",
            "midnightblue", "mintcream", "mistyrose",
            "moccasin", "navajowhite", "navy", "oldlace",
            "olive", "olivedrab", "orange", "orangered",
            "orchid", "palegoldenrod", "palegreen",
            "paleturquoise", "palevioletred", "papayawhip",
            "peachpuff", "peru", "pink", "plum",
            "powderblue", "purple", "red", "rosybrown",
            "royalblue", "saddlebrown", "salmon",
            "sandybrown", "seagreen", "seashell", "sienna",
            "silver", "skyblue", "slateblue", "slategray",
            "slategrey", "snow", "springgreen", "steelblue",
            "tan", "teal", "thistle", "tomato", "turquoise",
            "violet", "wheat", "white", "whitesmoke",
            "yellow", "yellowgreen"
        );

    public void start(Stage stage) {
        ListView<String> lv = new ListView<>(data);
        lv.setCellFactory(new Callback<ListView<String>,
                                       ListCell<String>>() {
            @Override
            public ListCell<String> call(ListView<String> l){
                return new ColorBox();
            }
        });

        stage.setScene(new Scene(lv, 320, 750));
        stage.setTitle("Predefined HTML/CSS Colors");
        stage.show();
    }

    private static class ColorBox extends ListCell<String> {
        @Override
        protected void updateItem(String col, boolean empty){
            super.updateItem(col, empty);
            if (empty || col == null) {
                setGraphic(null);
                setText(null);
            } else {
                Color c = Color.web(col);
                Rectangle rect = new Rectangle(100, 20);
                rect.setFill(c);
                setGraphic(rect);
                setText(col.toUpperCase());
                int r = (int)(255*c.getRed()),
                    g = (int)(255*c.getGreen()),
                    b = (int)(255*c.getBlue());
                setTooltip(new Tooltip(
                    col + ": (" + r + ", " + g + ", " + b +
                    ") = #" + String.format("%02X%02X%02X",
                                            r, g, b)
                ));
            }
        }
    }
}
```

The program displays a list of the predefined CSS/HTML named colors, as shown below:

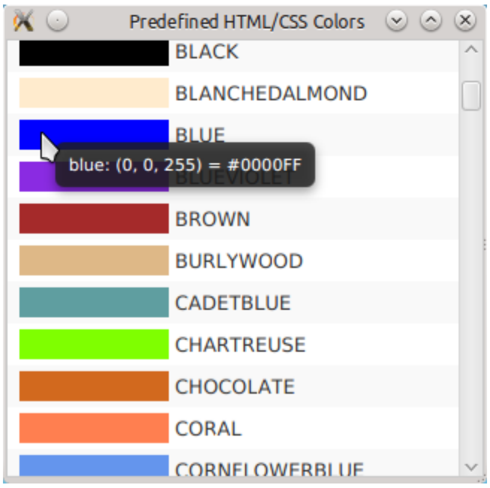


Table views are similar to list views. Let us consider the following example: rows of the table correspond to objects of class **PersonFX** and columns to properties of these objects:

Listing 149 JYN-TableFX/PersonFX.java

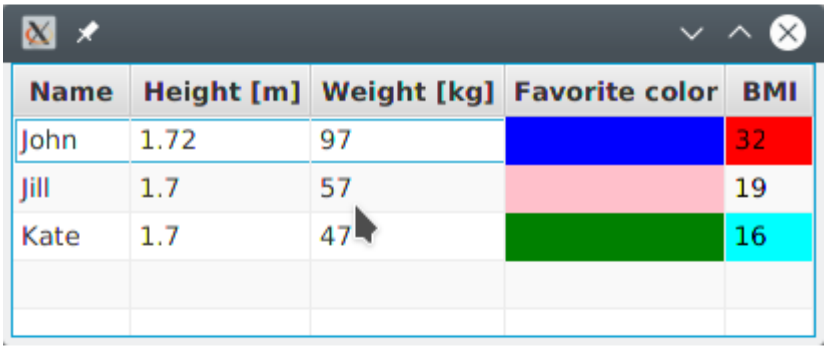
```java
import javafx.beans.property.ObjectProperty;
import javafx.beans.property.SimpleObjectProperty;
import javafx.beans.property.IntegerProperty;
import javafx.beans.property.SimpleIntegerProperty;
import javafx.beans.property.DoubleProperty;
import javafx.beans.property.SimpleDoubleProperty;
import javafx.beans.property.StringProperty;
import javafx.beans.property.SimpleStringProperty;
import javafx.scene.paint.Color;

public class PersonFX {
    private StringProperty name;
    private DoubleProperty height;
    private IntegerProperty weight;
    private ObjectProperty<Color> favCol;

    public PersonFX(String n, double h, int w, Color c) {
        name = new SimpleStringProperty(n);
        if (h > 3) h = h/100;
        height = new SimpleDoubleProperty(h);
        weight = new SimpleIntegerProperty(w);
        favCol = new SimpleObjectProperty<>(c);
    }

    public String getName() { return name.get(); }
    public double getHeight() { return height.get(); }
    public DoubleProperty heightProperty() {
        return height;
    }
    public int    getWeight() { return weight.get(); }
    public Color  getFavCol() { return favCol.get(); }
    public Integer getBmi() {
        return (int)(weight.get()/height.get()/height.get());
    }
}
```

and now we build the table itself:

Listing 150 JYN-TableFX/PersonTableFX.java

```java
import javafx.application.Application;
import javafx.collections.FXCollections;
import javafx.scene.Scene;
import javafx.scene.control.TableCell;
import javafx.scene.control.TableColumn;
import javafx.scene.control.TableView;
import javafx.scene.control.cell.PropertyValueFactory;
import javafx.scene.layout.Background;
import javafx.scene.layout.BackgroundFill;
import javafx.scene.layout.StackPane;
import javafx.scene.paint.Color;
import javafx.stage.Stage;

public class PersonTableFX extends Application {
    @Override
    public void start(Stage primaryStage) {
        TableView<PersonFX> persons =
            new TableView<>(FXCollections
                           .observableArrayList(
                new PersonFX("John", 1.72, 97, Color.BLUE),
                new PersonFX("Jill",  170, 57, Color.PINK),
                new PersonFX("Kate",  170, 47, Color.GREEN)
        ));


        TableColumn<PersonFX, String> nameCol =
                new TableColumn<>("Name");
        nameCol.setCellValueFactory(
                new PropertyValueFactory<>("name"));
        persons.getColumns().add(nameCol);

        TableColumn<PersonFX, Double> heightCol =
                new TableColumn<>("Height [m]");
        heightCol.setCellValueFactory(
                new PropertyValueFactory<>("height"));
        persons.getColumns().add(heightCol);

        TableColumn<PersonFX, Integer> weightCol =
                new TableColumn<>("Weight [kg]");
        weightCol.setCellValueFactory(
                new PropertyValueFactory<>("weight"));
        persons.getColumns().add(weightCol);

        TableColumn<PersonFX, Color> favColCol =
                new TableColumn<>("Favorite color");
        favColCol.setCellValueFactory(
                new PropertyValueFactory<>("favCol"));
        favColCol.setCellFactory(column -> {
            return new TableCell<PersonFX, Color>() {
                @Override
                protected void updateItem(Color col,
                                          boolean empty) {
                    super.updateItem(col, empty);

                    if (col == null || empty) {
                        setText(null);
                        setStyle("");
                        return;
                    }
                    setBackground(new Background(
                            new BackgroundFill(col,
                                               null, null)));
                }
            };
        });
        persons.getColumns().add(favColCol);

          // bmi is not a fx property, but this will work,
          // although it will be wrapped in
          // ReadOnlyObjectWrapper and cannot be observed
        TableColumn<PersonFX, Integer> bmiCol =
            new TableColumn<>("BMI");
        bmiCol.setCellValueFactory(
                new PropertyValueFactory<>("bmi"));
        bmiCol.setCellFactory(column -> {
            return new TableCell<PersonFX, Integer>() {
                @Override
                protected void updateItem(Integer bmi,
                                          boolean empty) {
                    super.updateItem(bmi, empty);

                    if (bmi == null || empty) {
                        setText(null);
                        setStyle("");
                        return;
                    }
                    setText(""+bmi);
                    if (bmi > 25) {
                        setTextFill(Color.BLACK);
                        setStyle("-fx-background-color:red");
                    } else if (bmi < 18) {
                        setTextFill(Color.BLACK);
                        setStyle("-fx-background-color:cyan");
                    } else {
                        setTextFill(Color.BLACK);
                    }
                }
            };
        });


        persons.getColumns().add(bmiCol);

        Scene scene = new Scene(new StackPane(persons));

        primaryStage.setScene(scene);
        primaryStage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

The program displays



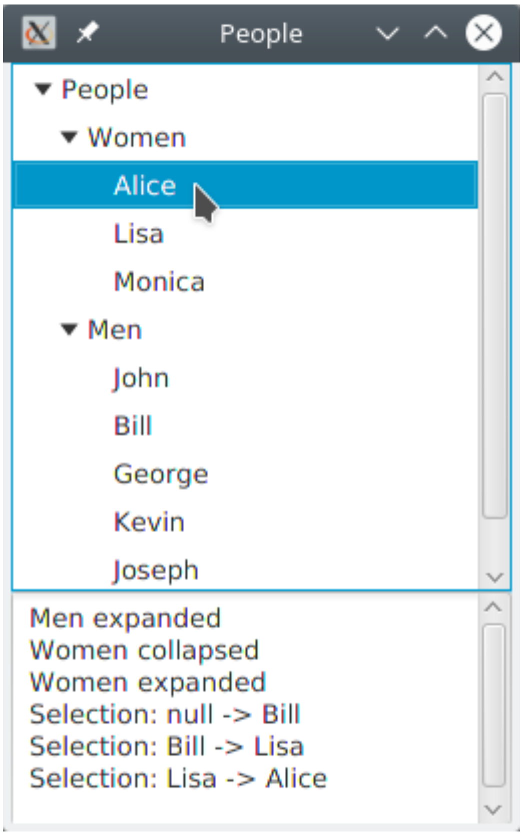
Name	Height [m]	Weight [kg]	Favorite color	BMI
John	1.72	97	🟦	32
Jill	1.7	57	■	19
Kate	1.7	47	🟩	16

Let us consider now trees, represented by objects of type **TreeView**.

Listing 151 JYO-TreeView/TreeFXBasic.java

```java
package tree;

import java.util.Arrays;
import java.util.List;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.TextArea;
import javafx.scene.control.TextField;
import javafx.scene.control.TreeCell;
import javafx.scene.control.TreeItem;
import javafx.scene.control.TreeView;
import javafx.scene.layout.VBox;
import javafx.scene.input.KeyCode;
import javafx.stage.Stage;

public class TreeFXBasic extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        TreeItem<String> treeRoot = new TreeItem<>("People");
        treeRoot.setExpanded(true);

        List<TreeItem<String>> women = Arrays.asList(
                new TreeItem<String>("Alice"),
                new TreeItem<String>("Lisa"),
                new TreeItem<String>("Monica")
                );
        TreeItem<String> womenNode = new TreeItem<>("Women");
        womenNode.getChildren().addAll(women);
        womenNode.setExpanded(true);
        treeRoot.getChildren().add(womenNode);

        List<TreeItem<String>> men = Arrays.asList(
                new TreeItem<String>("John"),
                new TreeItem<String>("Bill"),
                new TreeItem<String>("George"),
                new TreeItem<String>("Kevin"),
                new TreeItem<String>("Joseph")
                );
        TreeItem<String> menNode = new TreeItem<>("Men");
        menNode.getChildren().addAll(men);
        menNode.setExpanded(false);
        treeRoot.getChildren().add(menNode);

        TreeView<String> treeView = new TreeView<>();
        treeView.setRoot(treeRoot);

        TextArea area = new TextArea();
        area.setPrefRowCount(15);

          // detecting expand/collapse node events
        treeRoot.addEventHandler(
            TreeItem.<String>branchExpandedEvent(),
            e -> area.appendText(
                e.getSource().getValue() + " expanded\n"));
        treeRoot.addEventHandler(
            TreeItem.<String>branchCollapsedEvent(),
            e -> area.appendText(
                e.getSource().getValue() + " collapsed\n"));

          // editing nodes (on double click by default)
        treeView.setEditable(true);
        treeView.setCellFactory(view -> new MyTextCell());

          // detecting selections
        treeView.getSelectionModel().selectedItemProperty()
                .addListener( // ChangeListener
            (obsVal, oldV, newV) ->
                area.appendText("Selection: " +
                (oldV != null ? oldV.getValue() : "null") +
                " -> " + newV.getValue() + "\n")
        );

        VBox root = new VBox(treeView, area);
        Scene scene = new Scene(root,250,380);
        stage.setScene(scene);
        stage.setTitle("People");
        stage.show();
    }
}

class MyTextCell extends TreeCell<String> {
    private TextField textField;

    @Override
    public void startEdit() {
        super.startEdit(); // <- important
        textField = new TextField(getItem());
        textField.setOnKeyPressed(e -> {
            if (e.getCode() == KeyCode.ENTER)
                commitEdit(textField.getText());
            else if (e.getCode() == KeyCode.ESCAPE)
                cancelEdit();

            });
        setText(null);
        setGraphic(textField);
        textField.selectAll();
    }

    @Override
    public void cancelEdit() {
        super.cancelEdit();
          // restore normal ('label') view
        setText(getItem());
        setGraphic(getTreeItem().getGraphic());
    }

    @Override
    public void updateItem(String item, boolean empty) {
        super.updateItem(item, empty);

        if (empty) {
            setText(null);
            setGraphic(null);
            return;
        }

        if (!isEditing()) {
            setText(getItem());
            setGraphic(getTreeItem().getGraphic());
            return;
        }

        if (textField != null) {
            textField.setText(getItem());
        }
        setText(null);
        setGraphic(textField);
    }
}
```

The program displays a tree: nodes are editable (after double-clicking) and react to expanding/collapsing or selections.



People
Women
Alice
Lisa
Monica
Men
John
Bill
George
Kevin
Joseph

```
Men expanded
Women collapsed
Women expanded
Selection: null -> Bill
Selection: Bill -> Lisa
Selection: Lisa -> Alice
```

12.7 Tasks

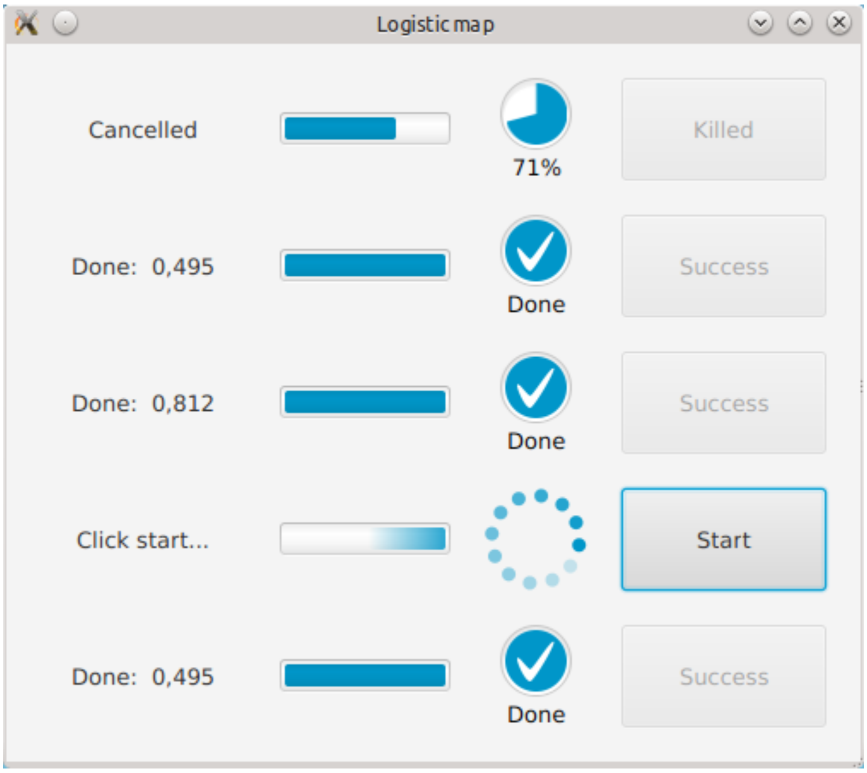
To avoid performing long tasks on the application thread (what makes the application irresponsive), we can create objects of type Task and execute them (their call method) on other threads. Tasks have many useful methods which provide means of communication between the task and the application running on the application thread. Changes of their public properties and change notifications for state, errors, and for event handlers, are executed on the application thread and are thread safe.

Let us consider an example

Listing 152 JXO-Tasks/Tasks.java

```java
package tasks;

import javafx.application.Application;
import javafx.application.Platform;
import javafx.concurrent.Task;
import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.stage.Stage;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.ProgressBar;
import javafx.scene.control.ProgressIndicator;
import javafx.scene.layout.ColumnConstraints;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.Priority;

public class Tasks extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        GridPane root = new GridPane();
        root.setHgap(20);
        root.setVgap(20);
        ColumnConstraints c1 = new ColumnConstraints();
        c1.setHgrow(Priority.ALWAYS);
        ColumnConstraints c2 = new ColumnConstraints(100);
        ColumnConstraints c3 = new ColumnConstraints(60);
        ColumnConstraints c4 = new ColumnConstraints();
        c4.setHgrow(Priority.ALWAYS);
        root.getColumnConstraints().addAll(c1, c2, c3, c4);
        root.setPadding(new Insets(20, 20, 20, 20));

          // adding elements to the grid
        for (int i = 0; i < 5; ++i) new OneRow(root, i);

        stage.setScene(new Scene(root));
        stage.setTitle("Logistic map");
        stage.show();
    }
}

class OneRow {
    final static int ITER = 100;
    GridPane grid;
    Label label = null;
    Button button = null;
    ProgressBar pBar = new ProgressBar();
    ProgressIndicator pInd = new ProgressIndicator();

    OneRow(GridPane g, int row){
        grid  = g;
        Task<Double> task = new Task<Double>() {
            int t = (int)(100+Math.random()*200);
            double d = Math.random();
            @Override
            protected Double call() {
                boolean cancelled = false;
                //label.textProperty()
                //     .bind(this.messageProperty());
                updateMessage("0/" + ITER);
                for (int i = 0; i <= ITER; ++i) {
                    if (isCancelled()) {
                        updateMessage("Cancelled");
                        cancelled = true;
                        break;
                    }
                    updateMessage(i + "/" + ITER);
                    updateProgress(i, ITER);
                    try {
                        Thread.sleep(t);
                    } catch (InterruptedException e) {
                        if (isCancelled()) {
                            updateMessage("Cancelled");
                            cancelled = true;
                            break;
                        }
                    }
                    d = 3.25*d*(1-d); // logistic map
                }
                if (!cancelled) {
                    updateProgress(ITER, ITER);
                    updateMessage("Done: " +
                        String.format("%6.3f",d));
                    Platform.runLater(() -> {
                        button.setDisable(true);
                        button.setText("Success");
                    });
                }
                return d;
            }
            @Override
            protected void running() {
                label.textProperty()
                     .bind(this.messageProperty());
            }
        };

        Thread thread = new Thread(task);
        thread.setDaemon(true);

        button = new Button("Start");
        button.setPrefSize(120, 60);
        button.setOnAction(e -> {
            if (button.getText().equals("Start")) {
                thread.start();
                Platform.runLater(() ->
                    button.setText("Cancel")
                );
            } else if (button.getText().equals("Cancel")) {
                task.cancel();
                Platform.runLater(() -> {
                    button.setDisable(true);
                    button.setText("Killed");
                });
            }
        });


        pBar.progressProperty()
            .bind(task.progressProperty());
        pInd.progressProperty()
            .bind(task.progressProperty());
        label = new Label("Click start...");
        label.setAlignment(Pos.CENTER);
        label.setPrefSize(120, 60);
          // label bound to message in running above

        grid.add(label,  0, row);
        grid.add(pBar,   1, row);
        grid.add(pInd,   2, row);
        grid.add(button, 3, row);
    }
}
```

which displays



Cancelled 71% Killed

Done: 0,495 Done Success

Done: 0,812 Done Success

Click start... Start

Done: 0,495 Done Success

12.8 *Animations*

Maybe the simplest way to create an animation in JavaFX is by using the abstract AnimationTimer class. The class itself has nothing to do with rendering any graphical interfaces — it inherits directly from **Object** and adds only three methods. Two of them are concrete (already implemented): **start** and **stop**. The third one, **handle**, is abstract and we have to implement it in our own, perhaps anonymous, class. After creation of an object of this class, one can call **start** on it. From this moment, the **handle** method will be called on the object many (around 50) times per second, until **stop** is invoked. Each time **handle** is called, a timestamp is passed to it as the only argument — it corresponds to time in nanoseconds ($1\,\mathrm{s} = 10^9\,\mathrm{ns}$).

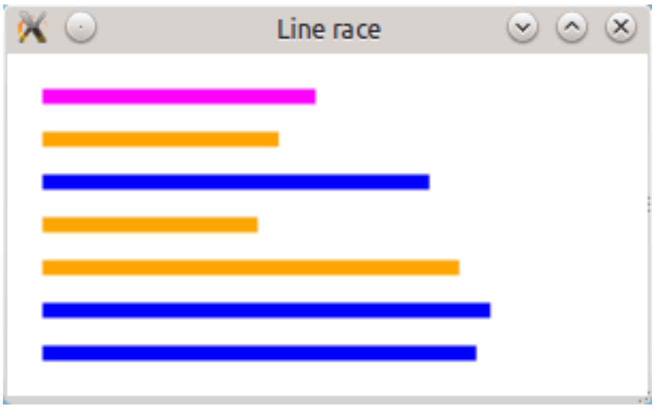
Let us look at the example below. We create (in the loop starting at line 24) several lines. With each of them, we associate an object of the **AnimationTimer** class. Its **handle** method calculates the time interval since the previous invocation (kept in prevT) and modifies the x-coordinate of the end point of the line: it is incremented by this time interval (in seconds) multiplied by the velocity v (which is generated randomly for each line separately). When the end point of the line reaches the width (minus some margin) of the pane within which the lines are rendered, the method **stop** is called and the animation stops:

Listing 153 JYP-AnimTimer/AnimTimer.java

```
 1 package animtimer;
 2 
 3 import javafx.animation.AnimationTimer;
 4 import javafx.application.Application;
 5 import javafx.beans.property.LongProperty;
 6 import javafx.beans.property.SimpleLongProperty;
 7 import javafx.scene.Scene;
 8 import javafx.scene.layout.Pane;
 9 import javafx.scene.paint.Color;
10 import javafx.scene.shape.Line;
11 import javafx.stage.Stage;
12 
13 public class AnimTimer extends Application {
14     private final static int
15         NLINES = 7, STEPY = 20, PREFWIDTH = 300, MARG = 20;
16     private final static Color[]
17         COLS = {Color.BLUE, Color.ORANGE, Color.MAGENTA};
18 
19     @Override
20     public void start(Stage stage) {
21         AnimationTimer[] timers = new AnimationTimer[NLINES];
22         Pane pane = new Pane();
23 
24         for (int i = 0; i < NLINES; ++i) {
25             Line line = new Line(MARG, (i+1)*STEPY,
26                                  MARG, (i+1)*STEPY);
27             line.setStroke(
28                     COLS[(int)(Math.random()*COLS.length)]);
29             line.setStrokeWidth(7);
30 
31             LongProperty prevT = new SimpleLongProperty(0);
32               // v is velocity in pixels/second
```

```
            double v = 10 + 20*Math.random();
            timers[i] = new AnimationTimer() {
                @Override
                public void handle(long t) {
                      // t is timestamp in nanoseconds
                    if (prevT.get() > 0) {
                        double dt = (t - prevT.get()) * 1e-9;
                        double newX = line.getEndX() + v*dt;
                        if (newX > pane.getWidth() - MARG) {
                            stop();
                            return;
                        }
                        line.setEndX(newX);
                    }
                    prevT.set(t);
                }
            };
            pane.getChildren().add(line);
        }
        pane.setPrefWidth(PREFWIDTH);
        pane.setPrefHeight(STEPY*(NLINES+1));
        stage.setScene(new Scene(pane));
        stage.setTitle("Line race");
        stage.show();

        for (int i = 0; i < NLINES; ++i) timers[i].start();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

The program displays



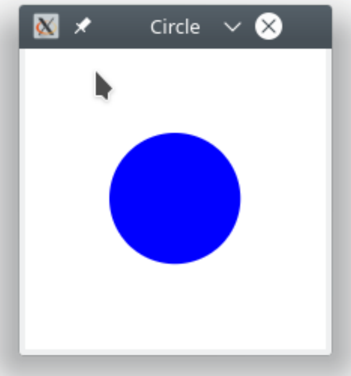
The next example illustrates the basic mechanisms that can be used to get animation effects — here, it is the radius of a circle.
The **Timeline** class describes the timeline, that is a series of frames each of which

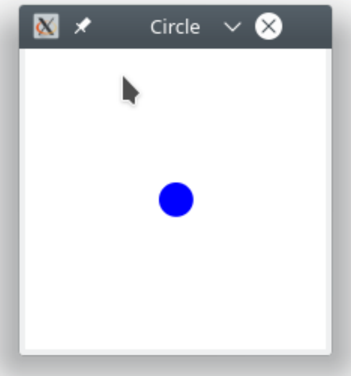
corresponds to a situation at one specified moment of time. By “situation” we mean the values of various properties of visible objects: radii of circles, x- and y-coordinates of points etc. It is described by an object of type KeyValue which can be constructed by passing a property (strictly speaking, a WritableValue) and its desired value. The third, optional, argument is an Interpolator — it specifies how the value of the property will be interpolated between the previous frame to the one described by this KeyValue (the default is Interpolator.LINEAR). Having defined one or more KeyValues corresponding to a given moment of time, we pass them to the constructor of the KeyFrame class. The first argument, of type Duration, specifies moment of time that this key frame corresponds to (counting from the beginning of the animation, *not* from the previous frame!). Finally, we create an object of type Timeline and add all desired KeyFrames to it.
The procedure is illustrated by the following simple example. Here, the radius of a Circle is the property which evolves in time:

Listing 154 JYR-BCirc/BreathingCircle.java

```java
package circle;

import javafx.animation.KeyFrame;
import javafx.animation.KeyValue;
import javafx.animation.Timeline;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.StackPane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.stage.Stage;
import javafx.util.Duration;

public class BreathingCircle extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        StackPane root = new StackPane();
        Circle circle = new Circle(10);
        circle.setFill(Color.BLUE);
        root.getChildren().add(circle);

        KeyValue rad0 = new KeyValue(circle.radiusProperty(),  10);
        KeyFrame fra0 = new KeyFrame(Duration.millis(0), rad0);

        KeyValue rad1 = new KeyValue(circle.radiusProperty(),  80);
        KeyFrame fra1 = new KeyFrame(Duration.millis(2000), rad1);

        KeyValue rad2 = new KeyValue(circle.radiusProperty(),  10);
        KeyFrame fra2 = new KeyFrame(Duration.millis(4000), rad2);

        Timeline timeline = new Timeline();
        timeline.getKeyFrames().addAll(fra0, fra1, fra2);
        timeline.setCycleCount(3);
        timeline.play();

        Scene scene = new Scene(root, 200, 200);
        stage.setScene(scene);
        stage.setTitle("Circle");
        stage.setResizable(false);
        stage.show();
    }
}
```

The program displays a “breathing” blue circle:





In some simple situations, when the value of a property just goes from a starting value to a final one, an implementation of the abstract Transition class can be used (it also extends the Animation class, as does Timeline). One can define a class extending Transition and provide an implementation of its abstract interpolate method, or use one its many concrete already defined implementations: they include

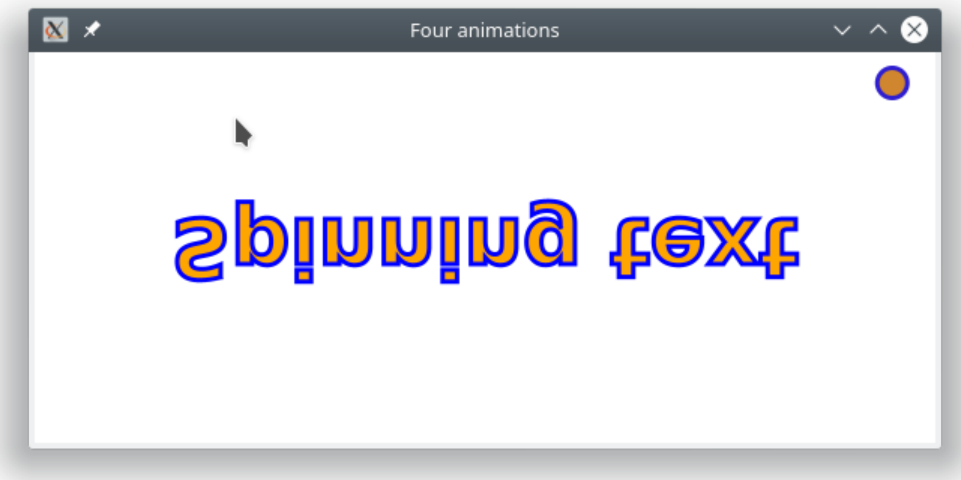
FadeTransition
FillTransition
RotateTransition
ScaleTransition
StrokeTransition
TranslateTransition

and others.
In the example below, we use three of them: ScaleTransition, FillTransition and StrokeTransition. In all cases, the first two arguments are duration and an object (Node or Shape, depending of the concrete class) whose property is evolving in time. Starting and final values can be set by appropriate methods, or, for some of these classes, also passed to the constructor. Note that we launch four animations (three transitions and one timeline) at the same time (lines 92-95) — the one defined by a timeline describes the movement of a little circle; its stroke and fill colors are gradually changed by appropriate transitions:

Listing 155 JYS-Trans/Trans.java

```java
package trans;

import javafx.animation.FillTransition;
import javafx.animation.KeyFrame;
import javafx.animation.KeyValue;
import javafx.animation.ScaleTransition;
import javafx.animation.StrokeTransition;
import javafx.animation.Timeline;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.StackPane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.scene.text.Font;
import javafx.scene.text.FontWeight;
import javafx.scene.text.Text;
import javafx.stage.Stage;
import javafx.util.Duration;

public class Trans extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        StackPane root = new StackPane();

        Text text = new Text("Spinning text");
        text.setStrokeWidth(3);
        text.setFill(Color.ORANGE);
        text.setStroke(Color.BLUE);
        text.setFont(Font.font(null, FontWeight.BOLD, 55));
        root.getChildren().addAll(text);
        ScaleTransition st = new ScaleTransition(
                Duration.millis(2000), text);
        st.setFromY(1);
        st.setToY(-1);
        st.setCycleCount(4);
        st.setAutoReverse(true);

        Circle circ = new Circle(10);
        circ.setStrokeWidth(3);
        root.getChildren().add(circ);
        FillTransition ft = new FillTransition(
                Duration.millis(8000), circ,
                Color.ORANGE, Color.BLUE);
        StrokeTransition tt = new StrokeTransition(
                Duration.millis(8000), circ,
                Color.BLUE, Color.ORANGE);

          // x and y relative to center, as this is StackPane
        KeyValue x0 = new KeyValue(
                circ.translateXProperty(), -280);
        KeyValue y0 = new KeyValue(
                circ.translateYProperty(), -110);
        KeyFrame f0 = new KeyFrame(
                Duration.millis(   0), x0, y0);

        KeyValue x1 = new KeyValue(
                circ.translateXProperty(), +280);
        KeyValue y1 = new KeyValue(
                circ.translateYProperty(), -110);
        KeyFrame f1 = new KeyFrame(
                Duration.millis(2000), x1, y1);

        KeyValue x2 = new KeyValue(
                circ.translateXProperty(), +280);
        KeyValue y2 = new KeyValue(
                circ.translateYProperty(), +110);
        KeyFrame f2 = new KeyFrame(
                Duration.millis(4000), x2, y2);

        KeyValue x3 = new KeyValue(
                circ.translateXProperty(), -280);
        KeyValue y3 = new KeyValue(
                circ.translateYProperty(), +110);
        KeyFrame f3 = new KeyFrame(
                Duration.millis(6000), x3, y3);

        KeyValue x4 = new KeyValue(
                circ.translateXProperty(), -280);
        KeyValue y4 = new KeyValue(
                circ.translateYProperty(), -110);
        KeyFrame f4 = new KeyFrame(
                Duration.millis(8000), x4, y4);

        Timeline timeline = new Timeline();
        timeline.getKeyFrames().addAll(f0, f1, f2, f3, f4);
        timeline.setCycleCount(1);

        st.play();
        ft.play();
        tt.play();
        timeline.play();

        stage.setScene(new Scene(root, 600, 260));
        stage.setTitle("Four animations");
        stage.show();
    }
}
```

The program displays



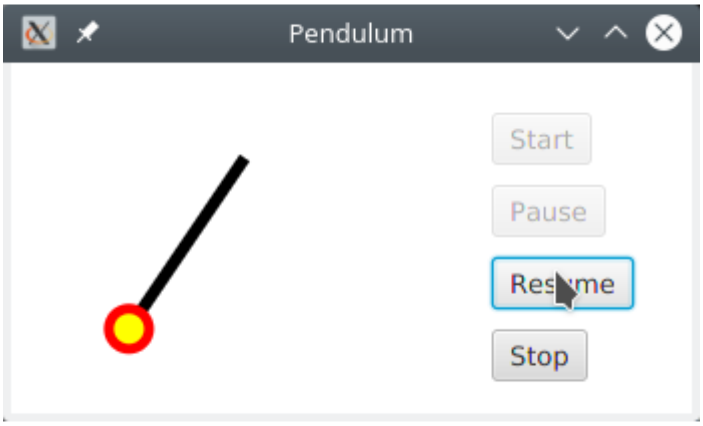
The last example displays a swinging pendulum. Its rod is represented by a **Line** and the weight by a **Circle**. One end of the rod is fixed. We then create a double property time, representing uniformly passing time. Two bindings, xBind and yBind depend on time and represent x- and y-coordinates of the center of the weight and of the second end of the rod (and are bound to these properties). Animation, defined by a timeline, consists of just two frames passed directly to the constructor: starting frame at time 0 and the final at time equal to one period of the pendulum. It is repeated indefinitely, though.

The buttons start, stop, pause and resume swinging — they use the appropriate methods of **Animation** class. We want some buttons to be disabled, when clicking them would not make sense. Therefore, we bind their disableProperty with appropriate logical values. For example, if the state of the animation is *not* STOPPED, then start button should be disabled. Similarly, the stop button should be disabled when the animation *is* already stopped.

Listing 156 JXR-AnimPendul/Pendulum.java

```
package pendulum;

import javafx.animation.Animation;
import javafx.animation.KeyFrame;
import javafx.animation.KeyValue;
import javafx.animation.Timeline;
import javafx.application.Application;
import javafx.application.Platform;
import javafx.beans.binding.DoubleBinding;
import javafx.beans.property.DoubleProperty;
import javafx.beans.property.SimpleDoubleProperty;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.VBox;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.scene.shape.Line;
import javafx.stage.Stage;
import javafx.util.Duration;

public class Pendulum extends Application {
    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage stage) {
        final double radius = 10, len = 100,
                     x0 = len+1.5*radius, y0 = 50,
                     period = 3, maxang = Math.PI/2.5,
                     omega = 2*Math.PI/period;

        DoubleProperty time = new SimpleDoubleProperty();

        Line line = new Line();
        line.setStartX(x0);
        line.setStartY(y0);
        line.setStrokeWidth(6);
        line.setStroke(Color.BLACK);

        Circle circle = new Circle(radius, Color.YELLOW);
        circle.setStrokeWidth(5);
        circle.setStroke(Color.RED);

        DoubleBinding xBind = new DoubleBinding() {
            {
                super.bind(time);
            }
            @Override
            protected double computeValue() {
                return x0 + len*Math.sin(
                    maxang*Math.sin(
                        omega*time.get() - Math.PI/2));
            }
        };
        DoubleBinding yBind = new DoubleBinding() {
            {
                super.bind(time);
            }
            @Override
            protected double computeValue() {
                return y0 + len*Math.cos(
                    maxang*Math.sin(
                        omega*time.get() - Math.PI/2));
            }
        };

        line.endXProperty().bind(xBind);
        line.endYProperty().bind(yBind);
        circle.centerXProperty().bind(xBind);
        circle.centerYProperty().bind(yBind);

        Timeline anim = new Timeline(
                  // KeyFrame at zero seems to be needed
                  // if we want to be able to stop animation
                  // and then start it from the beginning
                new KeyFrame(Duration.seconds(0),
                             new KeyValue(time, 0)),
                new KeyFrame(Duration.seconds(period),
                             new KeyValue(time, period)));
        anim.setCycleCount(Animation.INDEFINITE);

        Button startB = new Button("Start");
        startB.setOnAction(e -> anim.playFromStart());
        Button pauseB = new Button("Pause");
        pauseB.setOnAction(e -> anim.pause());
        Button resumeB = new Button("Resume");
        resumeB.setOnAction(e -> anim.play());
        Button stopB = new Button("Stop");
        stopB.setOnAction(e -> anim.stop());

          // disable buttons which should not
          // be active in a given situation
        startB.disableProperty().bind(
                anim.statusProperty()
               .isNotEqualTo(Animation.Status.STOPPED));
        pauseB.disableProperty().bind(
                anim.statusProperty()
               .isNotEqualTo(Animation.Status.RUNNING));
        resumeB.disableProperty().bind(
                anim.statusProperty()
               .isNotEqualTo(Animation.Status.PAUSED));
        stopB.disableProperty().bind(
                anim.statusProperty()
               .isEqualTo(Animation.Status.STOPPED));

        VBox buttons =
            new VBox(10, startB, pauseB, resumeB, stopB);
        buttons.relocate(2*len+4*radius, y0/2);
        Group group = new Group(line, circle, buttons);
        Scene scene = new Scene(group, 3*len+4*radius,
                                       y0+len+2.5*radius);
        stage.setScene(scene);
        stage.setTitle("Pendulum");
        stage.show();
    }
}
```

The GUI of the program looks like this:



12.9 JavaFX and CSS

All graphical components (nodes) may be styled by using CSS-like syntax (basically, just *property:value(s)* pairs). Styles can be specified by style sheets (external files — local, remote, or packed into *.jar* archives) or inline styles applied to individual nodes directly within the source code of the application.

For many JavaFX classes (in particular, those from *javafx.scene.control* package, but not from *javafx.scene.layout*), their names can be used as CSS style-class names and may be used as CSS selectors; CSS property names are formed from JavaFX variables (properties) names. There is a convention that names used in CSS are always all lowercase with subwords separated by a hyphen. So, for example, JavaFX class RadioButton corresponds to CSS style-class radio-button. For CSS property names the rule is similar, but the prefix `-fx-` is added: therefore, the JavaFX property textAlignment will correspond to CSS property name -fx-text-alignment.

There are very many JavaFX classes, objects of which can be styled, and for each of them there are many properties that can be assigned various values. The full list can be found at

```
https://docs.oracle.com/javase/10/docs/api/
        javafx/scene/doc-files/cssref.html
```

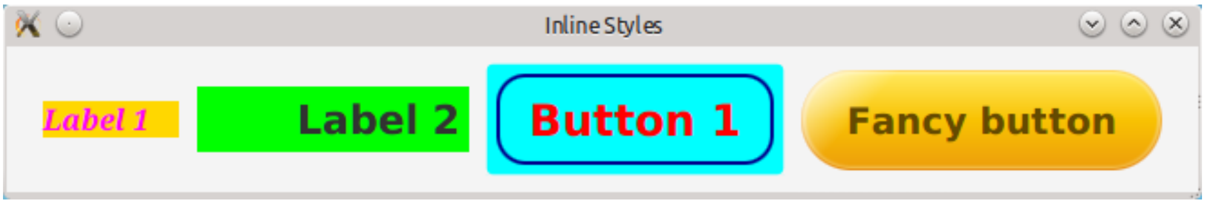
Below, we will show just a few examples — the details should be looked up in the documentation.

Let's start with an example of styling graphical components using inlined CSS. This is quite simple: on a node, we call the method **setStyle** passing, as a String, one or more semicolon separated ***property:value(s)*** pairs. The style will be applied to the object in question only; hence, there is no need for any selectors here. Of course, all properties must be valid properties of this particular type of node; also assigned values must conform to the specification (see the aforementioned documentation).

Listing 157 JYA-InlineStyles/InlineStyles.java

```java
package inlinestyles;

// see https://docs.oracle.com/javase/10/docs/api/
//             javafx/scene/doc-files/cssref.html

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

public class InlineStyles extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        HBox root = new HBox();
        root.setStyle(
            "-fx-spacing:        10px;" +
            "-fx-fill-height:   false;" +
            "-fx-alignment:    center;" +
            "-fx-padding: 10 20 10 20;"
        );

        Label l1 = new Label("Label 1");
        l1.setStyle(
            "-fx-font-family:     serif;" +
            "-fx-font-size:        16px;" +
            "-fx-font-style:    oblique;" +
            "-fx-font-weight:      bold;" +
            "-fx-pref-width:         75;" +
            "-fx-background-color: gold;" +
            "-fx-text-fill:     fuchsia;"
        );
        Label l2 = new Label("Label 2");
        l2.setStyle(
            "-fx-font-family: sans-serif;" +
            "-fx-font-size:         22px;" +
            "-fx-font-style:      normal;" +
            "-fx-font-weight:       bold;" +
            "-fx-pref-width:         150;" +
            "-fx-background-color:  lime;" +
            "-fx-padding: 5 0 5 55;"
        );

        Button b1 = new Button("Button 1");
        b1.setStyle(
            "-fx-font-size:            24px;" +
            "-fx-font-weight:          bold;" +
            "-fx-background-color:     aqua;" +
            "-fx-text-fill:             red;" +
            "-fx-border-style: solid inside;" +
            "-fx-border-width:            2;" +
            "-fx-border-insets:           5;" +
            "-fx-border-radius:          15;" +
            "-fx-border-color:     darkblue;"
        );

        Button b2 = new Button("Fancy button");
        b2.setStyle(
              // designed by Jasper Potts
            "-fx-background-color: " +
                "linear-gradient(#ffd65b, #e68400)," +
                "linear-gradient(#ffef84, #f2ba44)," +
                "linear-gradient(#ffea6a, #efaa22)," +
                "linear-gradient(#ffe657 0%, #f8c202" +
                " 50%, #eea10b 100%)," +
                "linear-gradient(from 0% 0% to 15% 50%," +
                " rgba(255,255,255,0.9)," +
                " rgba(255,255,255,0));" +
            "-fx-background-radius: 30;" +
            "-fx-background-insets: 0,1,2,3,0;" +
            "-fx-text-fill: #654b00;" +
            "-fx-font-weight: bold;" +
            "-fx-font-size: 20px;" +
            "-fx-padding: 15 25 15 25;"
        );

        root.getChildren().addAll(l1, l2, b1, b2);
        stage.setScene(new Scene(root));
        stage.setTitle("Inline Styles");
        stage.show();
    }
}
```

The program displays



Next example is a bit more involved. Here , we change the look of our UI dynamically, at run-time. We will use both inlined styles and styles defined in two different, although similar, external style-sheet files: ***sheet1.css***

Listing 158 JYB-DynamicStyles/sheet1.css

```
.root {
     -fx-bkg: azure;
     -fx-background-color: -fx-bkg;
}
.label {
     -fx-text-fill: blue;
}
.button {
     -fx-text-fill: blue;
     -fx-font: normal bold 12px serif;
}
.vbox {
     -fx-padding:       5 5 15 5;
     -fx-alignment:     top-center;
     -fx-border-color: black;
     -fx-border-style: solid;
     -fx-border-width: 2;
}
.vbox .label {
     -fx-text-fill: orange;
}
.hbox {
     -fx-spacing:    15;
     -fx-alignment: center;
     -fx-padding:    10 20 10 20;
}
```

and ***sheet2.css***

Listing 159 JYB-DynamicStyles/sheet2.css

```
.root {
     -fx-bkg: lightyellow;
     -fx-background-color: -fx-bkg;
}
.label {
     -fx-text-fill: red;
}
.button {
    -fx-text-fill: red;
    -fx-font: normal normal 12px sans-serif;
}
.vbox {
    -fx-padding:      5 5 15 5;
    -fx-alignment:    top-center;
    -fx-border-color: red;
    -fx-border-style: dashed;
    -fx-border-width: 2;
}
.vbox .label {
    -fx-text-fill: orange;
}
.hbox {
    -fx-spacing:   15;
    -fx-alignment: center;
    -fx-padding:   10 20 10 20;
}
```

External style sheets are set on the scene object (see line 74 of the program below). The .root selector matches the root element of the scene; due to inheritance you can put here properties that are to be inherited by root's children. In particular, you can define CSS 'variables' here. We do it in both CSS sheets, for example in ***sheet1.css***

```
.root {
    -fx-bkg: azure;
    -fx-background-color: -fx-bkg;
}
```

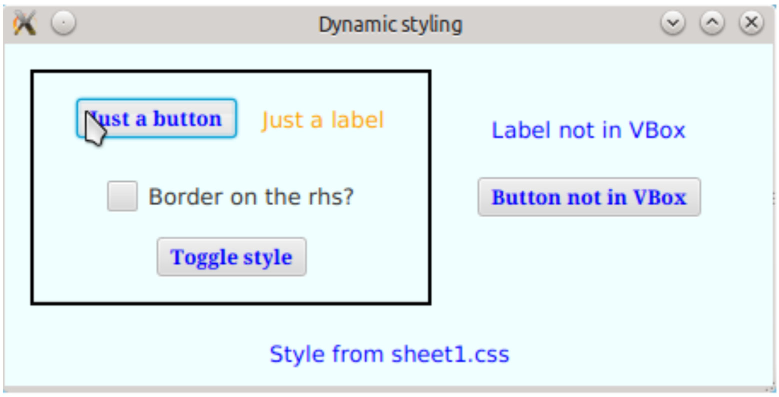
Note that -fx-bkg does not correspond to any property: we just introduce a 'variable' with this name and assign a value (in this case color azure) to it. Then we set the color of the background to the value of this variable. The advantage of this approach is the fact that this variable with its value is inherited by the root's children and we can use it to set styling of other nodes. For example, look at line 64 of the program: there, we assign a color -fx-bkg to the -fx-border-color property, so it will be the same as the color of the background, whatever it currently is (to make the border invisible). In the present version of JavaFX such variables seem to work only for colors and also sizes, but without units (px will be assumed).

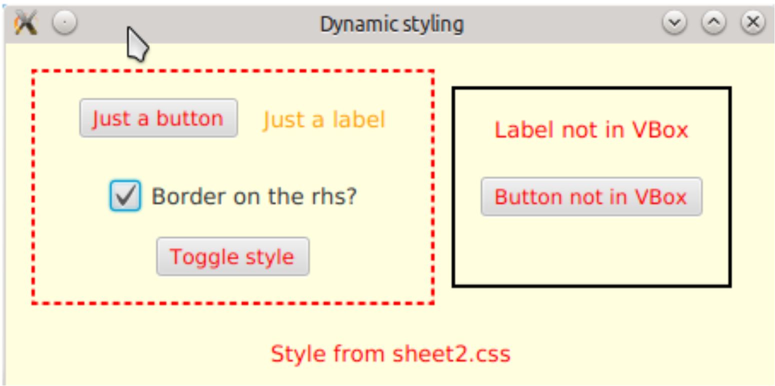
Listing 160 JYB-DynamicStyles/DynamicStyles.java

```
1  package dynamicstyles;
2
3  import javafx.application.Application;
4  import javafx.geometry.Insets;
5  import javafx.geometry.Pos;
6  import javafx.scene.Scene;
7  import javafx.scene.control.Button;
```

```java
import javafx.scene.control.CheckBox;
import javafx.scene.control.Label;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.FlowPane;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class DynamicStyles extends Application {
    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(final Stage stage) {
        String style1 = "sheet1.css";
        String style2 = "sheet2.css";

        HBox buttons = new HBox();
          // HBox style-class empty by default, so...
        buttons.getStyleClass().add("hbox");

        Button but = new Button("Just a button");
        Label  lab = new Label("Just a label");
        buttons.getChildren().addAll(but, lab);

        CheckBox brd = new CheckBox("Border on the rhs?");
        brd.setAllowIndeterminate(false); // default anyway
        brd.setSelected(false);

        Button toggle = new Button("Toggle style");

        VBox vbox = new VBox(15);
          // VBox style-class empty by default, so...
        vbox.getStyleClass().add("vbox");
        vbox.getChildren().addAll(buttons, brd, toggle);

        Label message = new Label("Style from " + style1);
        message.setPadding(new Insets(20, 10, 10, 10));
        BorderPane.setAlignment(message, Pos.CENTER);

        BorderPane bdPane = new BorderPane();
        bdPane.setPadding(new Insets(15));
        bdPane.setCenter(vbox);
        bdPane.setBottom(message);

        String withBorder =
            "-fx-orientation: vertical;" +
            "-fx-vgap: 20px;" +
            "-fx-column-halignment: center;" +
            "-fx-padding: 15;" +
            "-fx-border-color: black;" +
            "-fx-border-width: 2px;" +
            "-fx-border-insets: 10;" +
            "-fx-border-style: solid;";
        String noBorder = withBorder
                         .replace("black", "-fx-bkg");

        FlowPane fp = new FlowPane();
        fp.setStyle(noBorder);
        Label  labvb = new Label("Label not in VBox");
        Button butvb = new Button("Button not in VBox");
        fp.getChildren().addAll(labvb, butvb);
        bdPane.setRight(fp);

        Scene scene = new Scene(bdPane, 450, 200);
        scene.getStylesheets().add(style1);

        toggle.setOnAction(e -> {
            String curr =
                scene.getStylesheets().contains(style1) ?
                                       style2 : style1;
            scene.getStylesheets().clear();
            scene.getStylesheets().add(curr);
            message.setText("Style from " + curr);
        });

        brd.setOnAction(e -> {
            if (brd.isSelected()) fp.setStyle(withBorder);
            else                  fp.setStyle(noBorder);
        });

        stage.setTitle("Dynamic styling");
        stage.setScene(scene);
        stage.show();
    }
}
```

The program displays



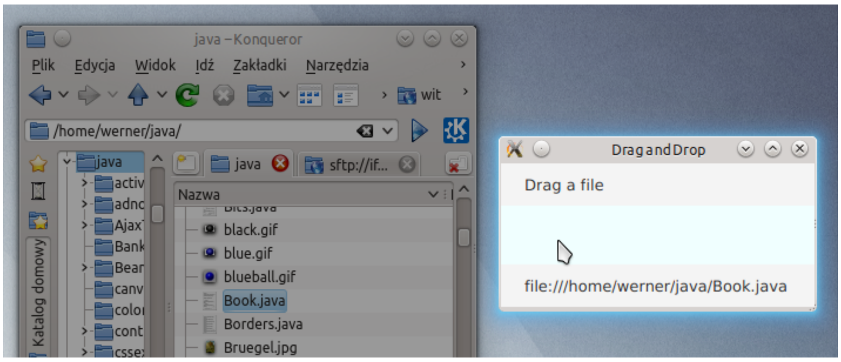


12.10 Drag and Drop

Listing 161 JYD-DnDFile/DnDFileName.java

```java
package dndfile;

import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.input.Dragboard;
import javafx.scene.input.DragEvent;
import javafx.scene.input.TransferMode;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class DnDFileName extends Application {
    @Override
    public void start(Stage stage) {
        Label prompt = new Label("Drag a file");
        prompt.setPadding(new Insets(10, 20, 10, 20));
        Label info   = new Label();
        info.setPadding(new Insets(10, 20, 10, 20));

        BorderPane bdPane = new BorderPane();
        bdPane.setTop(prompt);
        bdPane.setBottom(info);
        StackPane stack = new StackPane();
        stack.setStyle(
            "-fx-pref-width:         450;" +
            "-fx-max-width:          999;" +
            "-fx-pref-height:         50;" +
            "-fx-background-color: azure;"
        );
        bdPane.setCenter(stack);

        stack.addEventHandler(DragEvent.DRAG_OVER,
            e -> {
                if (e.getGestureSource() != bdPane &&
                    e.getDragboard().hasString()) {
                    e.acceptTransferModes(
                            TransferMode.COPY_OR_MOVE);
                }
                e.consume();
            }
        );
        stack.addEventHandler(DragEvent.DRAG_DROPPED,
            e -> {
                Dragboard db = e.getDragboard();
                boolean success = false;
                if (db.hasString()) {
                    info.setText(db.getString());
                    success = true;
                }
                  // nobody listens anyway
                e.setDropCompleted(success);
                e.consume();
            }
        );

        stage.setTitle("Drag and Drop");
        stage.setScene(new Scene(bdPane));
        stage.show();
    }
    public static void main(String[] args) {
        launch(args);
    }
}
```

The program displays



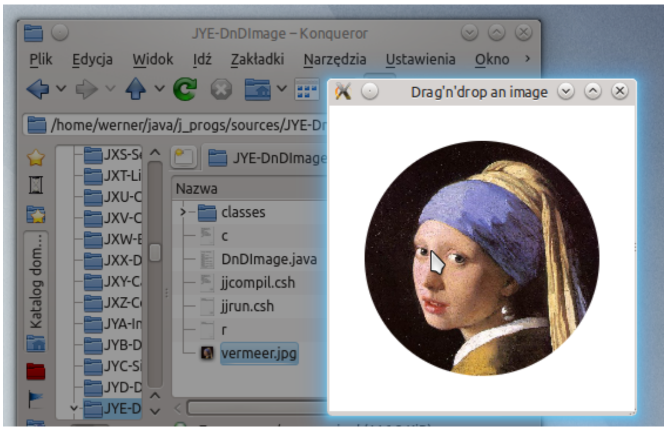
Drag a file

file:///home/werner/java/Book.java

Listing 162 JYE-DnDImage/DnDImage.java

```java
package dndimage;

import java.io.File;
import javafx.application.Application;
import javafx.scene.Node;
import javafx.scene.Scene;
import javafx.scene.layout.StackPane;
import javafx.scene.image.Image;
import javafx.scene.input.Dragboard;
import javafx.scene.input.DragEvent;
import javafx.scene.input.TransferMode;
import javafx.scene.paint.Color;
import javafx.scene.paint.ImagePattern;
import javafx.scene.shape.Circle;
import javafx.stage.Stage;

public class DnDImage extends Application {
    @Override
    public void start(Stage stage) {
        StackPane root = new StackPane();
        Circle circ = new Circle(130, 130, 100, Color.GRAY);
        root.getChildren().add(circ);

        circ.addEventHandler(DragEvent.DRAG_OVER,
            e ->  {
                Dragboard db = e.getDragboard();
                if(db.hasFiles())
                    e.acceptTransferModes(TransferMode.ANY);
                e.consume();
            }
        );

        circ.addEventHandler(DragEvent.DRAG_DROPPED,
            e -> {
                Dragboard dboard = e.getDragboard();
                boolean success = false;
                if (dboard.hasFiles()) {
                    File f = dboard.getFiles().get(0);
                    String path = f.toURI().toString();
                    circ.setFill(new ImagePattern(
                        new Image(path), 0, 0, 1, 1, true));
                    success = true;
                }
                e.setDropCompleted(success);
                e.consume();
            }
        );

        stage.setTitle("Drag'n'drop an image");
        stage.setScene(new Scene(root, 260, 260));
        stage.show();
    }
    public static void main(String[] args) {
        launch(args);
    }
}
```

The program displays

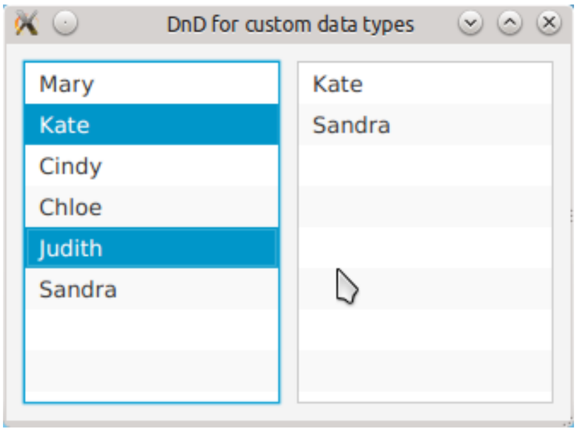


Listing 163 JYF-DnDLists/DnDLists.java

```
package dndlists;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;
import javafx.application.Application;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.geometry.Insets;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.ListView;
```

```java
import javafx.scene.input.ClipboardContent;
import javafx.scene.input.DataFormat;
import javafx.scene.input.Dragboard;
import javafx.scene.input.DragEvent;
import javafx.scene.input.MouseEvent;
import javafx.scene.input.TransferMode;
import javafx.scene.layout.FlowPane;
import javafx.stage.Stage;
import static javafx.scene.control.SelectionMode.MULTIPLE;

public class DnDLists extends Application {
      // our own data format, name irrelevant
    static final DataFormat PERS_FORM =
                            new DataFormat("PersonFormat");

    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage stage) {
        ObservableList<Person> persons =
            FXCollections.observableArrayList(
                new Person("Mary"), new Person("Kate"),
                new Person("Cindy"), new Person("Chloe"),
                new Person("Judith"), new Person("Sandra"));
        ListView<Person> listL = new ListView<>(persons);
        ListView<Person> listR = new ListView<>();

        FlowPane pane = new FlowPane(listL, listR);
        pane.setHgap(10);
        pane.setPadding(new Insets(10));

        for (ListView<Person> list :
                            Arrays.asList(listL, listR)) {
            list.getSelectionModel()
                .setSelectionMode(MULTIPLE);
            list.setPrefSize(150, 200);
              // DRAG DETECTED is a Mouse, not Drag event!
            list.addEventHandler(MouseEvent.DRAG_DETECTED,
                    e -> dragDetected(e, list));

            list.addEventHandler(DragEvent.DRAG_OVER,
                    e -> dragOver(e, list));

            list.addEventHandler(DragEvent.DRAG_DROPPED,
                    e -> dragDropped(e, list));

            list.addEventHandler(DragEvent.DRAG_DONE,
                    e -> dragDone(e, list));
        }

        stage.setScene(new Scene(pane, 330, 220));
        stage.setTitle("DnD for custom data types");
        stage.show();
    }

    private void dragDetected(MouseEvent e,
                              ListView<Person> lv) {
        if (lv.getSelectionModel()
              .getSelectedIndices().size() > 0) {
            Dragboard dBoard = lv.startDragAndDrop(
                                TransferMode.COPY_OR_MOVE);
            ArrayList<Person> selected = new ArrayList<>(
                lv.getSelectionModel().getSelectedItems());
            ClipboardContent cont = new ClipboardContent();
            cont.put(PERS_FORM, selected);
            dBoard.setContent(cont);
        }
        e.consume();
    }

    private void dragOver(DragEvent e, ListView<Person> lv) {
        Dragboard dragboard = e.getDragboard();
        if (e.getGestureSource() != lv &&
                dragboard.hasContent(PERS_FORM)) {
            e.acceptTransferModes(TransferMode.COPY_OR_MOVE);
        }
        e.consume();
    }

    @SuppressWarnings("unchecked")
    private void dragDropped(DragEvent e,
                             ListView<Person> lv) {
        boolean success = false;
        Dragboard dragboard = e.getDragboard();
        if(dragboard.hasContent(PERS_FORM)) {
              // casting here cannot be checked by compiler,
              // so we suppressed warnings for this method
            ArrayList<Person> list =
                (ArrayList<Person>)
                        dragboard.getContent(PERS_FORM);
            lv.getItems().addAll(
                list.stream()
                    .filter(p -> !lv.getItems().contains(p))
                    .collect(Collectors.toList())
            );
            success = true;
        }
        e.setDropCompleted(success);
        e.consume();
    }

    private void dragDone(DragEvent e, ListView<Person> lv) {
        if (e.getTransferMode() == TransferMode.MOVE) {
            List<Person> selected = new ArrayList<>(
                lv.getSelectionModel().getSelectedItems());
            lv.getSelectionModel().clearSelection();
            lv.getItems().removeAll(selected);
        }
        e.consume();
    }
}

class Person implements Serializable {
    private String name;
    public Person(String name) { this.name = name; }
    public String getName()    { return name;      }
    @Override
    public String toString()   { return name;      }
    @Override
    public boolean equals(Object other) {
        if (other == null || getClass() != other.getClass())
            return false;
        return (this == other ||
                name.equals(((Person)other).name));
    }
    @Override                   // not necessary here, but if
    public int hashCode() {     // equals is overriden, then
        return name.hashCode(); // hashCode should also be
    }
}
```

The program displays



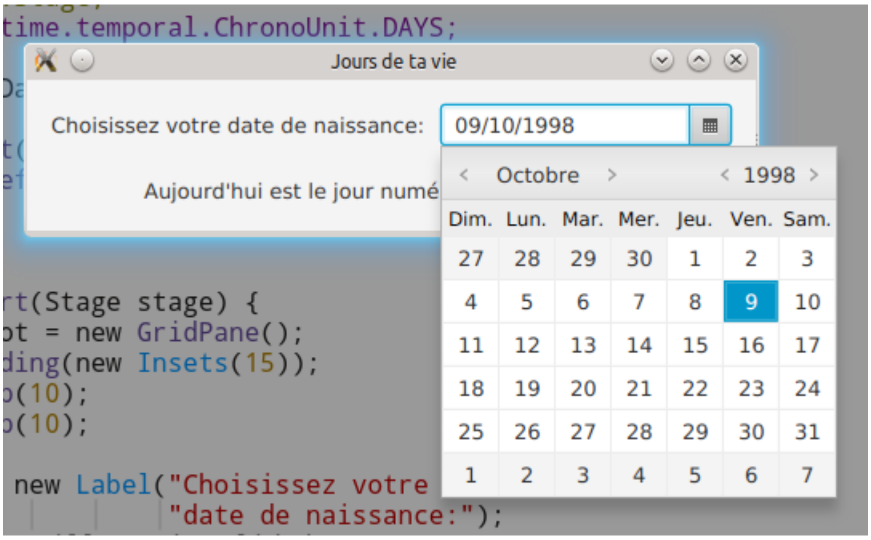
12.11 *Miscellanea*

Date picker:

Listing 164 JXS-SelectDate/SelectDate.java

```java
package seldate;

import java.time.LocalDate;
import java.util.Locale;
import javafx.application.Application;
import javafx.geometry.HPos;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.control.DatePicker;
import javafx.scene.control.Label;
import javafx.scene.layout.GridPane;
import javafx.stage.Stage;
import static java.time.temporal.ChronoUnit.DAYS;

public class SelectDate extends Application {
    @Override
    public void init() {
        Locale.setDefault(Locale.FRENCH);
    }

    @Override
    public void start(Stage stage) {
        GridPane root = new GridPane();
        root.setPadding(new Insets(15));
        root.setHgap(10);
        root.setVgap(10);

        Label lab = new Label("Choisissez votre " +
                              "date de naissance:");
          // so text will not be elided
        lab.setMinSize(Label.USE_PREF_SIZE,
                       Label.USE_PREF_SIZE);
        Label message = new Label();
        message.setPadding(new Insets(10));
        GridPane.setHalignment(message, HPos.CENTER);
        DatePicker picker = new DatePicker();
        root.add(lab,     0, 0);
        root.add(picker,  1, 0);
        root.add(message, 0, 1, 2, 1);
        picker.setOnAction(e -> {
            LocalDate date = picker.getValue();
            long days = DAYS.between(date, LocalDate.now());
            message.setText("Aujourd'hui est le jour " +
                            "num\u00e9ro " +
                            days + " de ta vie...");
        });
        stage.setTitle("Jours de ta vie");
        stage.setScene(new Scene(root));
        stage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

which produces



```
time.temporal.ChronoUnit.DAYS;
rt(Stage stage) {
pt = new GridPane();
ding(new Insets(15));
p(10);
p(10);
 new Label("Choisissez votre
            "date de naissance:");
```

Jours de ta vie

Choisissez votre date de naissance: 09/10/1998

Aujourd'hui est le jour numé

‹ Octobre › ‹ 1998 ›

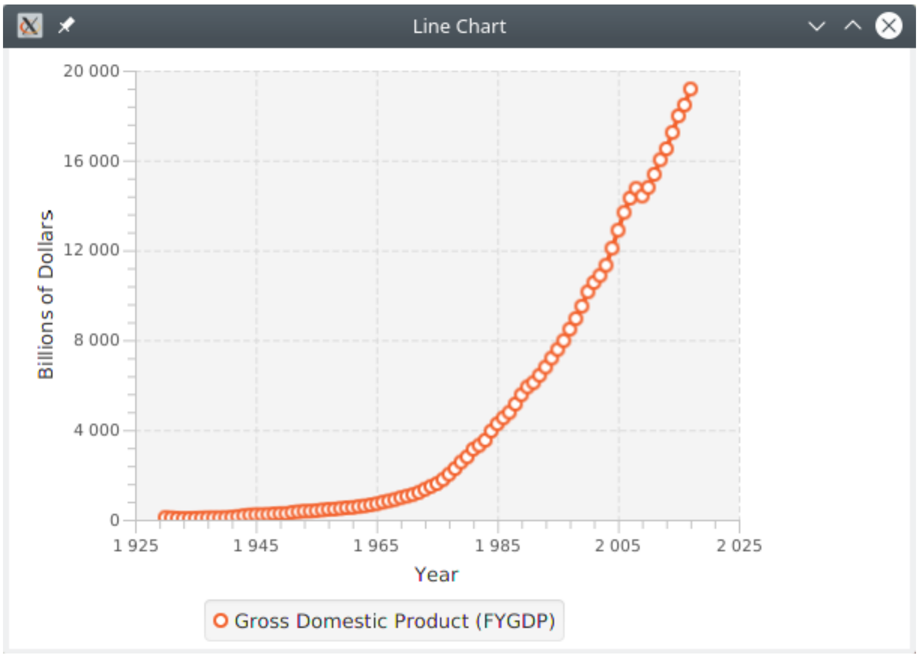
Dim.	Lun.	Mar.	Mer.	Jeu.	Ven.	Sam.
27	28	29	30	1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31
1	2	3	4	5	6	7

Linear chart:

Listing 165 JXT-LinChart/ChartExample.java

```java
package  chart;

import java.io.BufferedReader;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;

import javafx.application.Application;
import javafx.application.Platform;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.stage.Stage;
import javafx.scene.chart.LineChart;
import javafx.scene.chart.NumberAxis;
import javafx.scene.chart.ValueAxis;
import javafx.scene.chart.XYChart;

public class ChartExample extends Application {
    public static void main(String args[]){
        launch(args);
    }

    @Override
    public void start(Stage stage) {
        XYChart.Series<Number,Number> series =
                new XYChart.Series<>();
        series.setName("Gross Domestic Product (FYGDP)");
        try (BufferedReader br =
                Files.newBufferedReader(
                    Paths.get("grossProduct.csv"))) {
            String line;
            while ((line = br.readLine()) != null) {
                String[] s = line.split(",");
                series.getData().add(
                    new XYChart.Data<Number,Number>(
                            Integer.valueOf(s[0]),
                            Double.valueOf(s[1])));
            }
        } catch(IOException e) {
            System.out.println("No file or wrong format");
            Platform.exit();
        }

        NumberAxis xAxis =
                new NumberAxis("Year", 1925, 2025, 20);
        NumberAxis yAxis = new NumberAxis(0, 20000, 4000);
        yAxis.setLabel("Billions of Dollars");
        LineChart<Number,Number> chart =
                new LineChart<Number,Number>(xAxis, yAxis);
        chart.getData().add(series);

        Group root = new Group(chart);
        stage.setTitle("Line Chart");
        stage.setScene(new Scene(root, 600, 400));
        stage.show();
    }
}
```

which produces



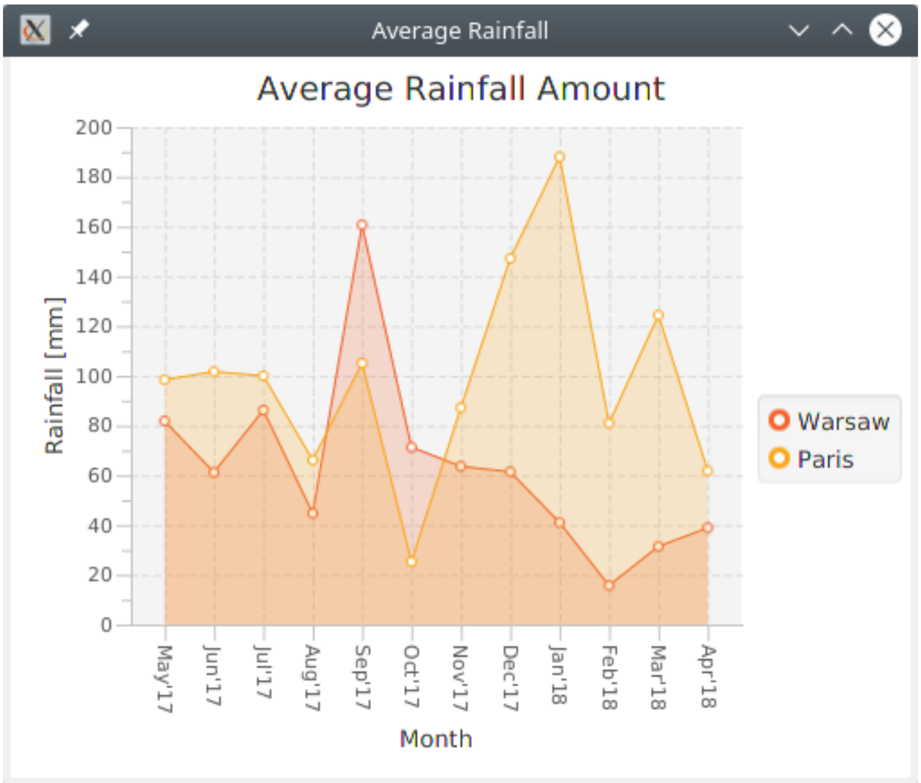
Category chart:

Listing 166 JXU-CatChart/CategoryChart.java

```java
package catchart;

import javafx.application.Application;
import javafx.geometry.Side;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.chart.AreaChart;
import javafx.scene.chart.CategoryAxis;
import javafx.scene.chart.NumberAxis;
import javafx.scene.chart.XYChart;
import javafx.stage.Stage;

public class CategoryChart extends Application {

    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage primaryStage) {
        CategoryAxis xAxis = new CategoryAxis();
        NumberAxis yAxis = new NumberAxis(0, 200, 20);
        xAxis.setLabel("Month");
        yAxis.setLabel("Rainfall [mm]");
        yAxis.setMinorTickCount(2); // 2 subintervals, 1 tic

          // Warsaw
        XYChart.Series<String,Number>
                seriesW = new XYChart.Series<>();
        seriesW.setName("Warsaw");
        seriesW.getData().add(
          new XYChart.Data<String,Number>("May'17",81.8));
        seriesW.getData().add(
          new XYChart.Data<String,Number>("Jun'17",61.2));
        seriesW.getData().add(
          new XYChart.Data<String,Number>("Jul'17",86.2));
        seriesW.getData().add(
          new XYChart.Data<String,Number>("Aug'17",44.7));
        seriesW.getData().add(
          new XYChart.Data<String,Number>("Sep'17",160.7));
        seriesW.getData().add(
          new XYChart.Data<String,Number>("Oct'17", 71.3));
        seriesW.getData().add(
          new XYChart.Data<String,Number>("Nov'17",63.6));
        seriesW.getData().add(
          new XYChart.Data<String,Number>("Dec'17",61.4));
        seriesW.getData().add(
          new XYChart.Data<String,Number>("Jan'18",41.0));
        seriesW.getData().add(
          new XYChart.Data<String,Number>("Feb'18",15.7));
        seriesW.getData().add(
          new XYChart.Data<String,Number>("Mar'18",31.4));
        seriesW.getData().add(
          new XYChart.Data<String,Number>("Apr'18",38.9));

         // Paris
        XYChart.Series<String,Number>
            seriesP = new XYChart.Series<>();
        seriesP.setName("Paris");
        seriesP.getData().add(
          new XYChart.Data<String,Number>("May'17",98.4));
        seriesP.getData().add(
          new XYChart.Data<String,Number>("Jun'17",101.7));
        seriesP.getData().add(
          new XYChart.Data<String,Number>("Jul'17",100.0));
        seriesP.getData().add(
          new XYChart.Data<String,Number>("Aug'17",66.1));
        seriesP.getData().add(
          new XYChart.Data<String,Number>("Sep'17",105.1));
        seriesP.getData().add(
          new XYChart.Data<String,Number>("Oct'17",25.3));
        seriesP.getData().add(
          new XYChart.Data<String,Number>("Nov'17",87.2));
        seriesP.getData().add(
          new XYChart.Data<String,Number>("Dec'17",147.2));
        seriesP.getData().add(
          new XYChart.Data<String,Number>("Jan'18",188.0));
        seriesP.getData().add(
          new XYChart.Data<String,Number>("Feb'18",80.9));
```

```
    seriesP.getData().add(
      new XYChart.Data<String,Number>("Mar'18",124.4));
    seriesP.getData().add(
      new XYChart.Data<String,Number>("Apr'18",61.8));

    AreaChart<String,Number>
        areaChart = new AreaChart<>(xAxis,yAxis);
    areaChart.setTitle("Average Rainfall Amount");
    areaChart.setLegendSide(Side.RIGHT);

    areaChart.getData().add(seriesW);
    areaChart.getData().add(seriesP);

    Group root = new Group();
    root.getChildren().add(areaChart);
    primaryStage.setTitle("Average Rainfall");
    primaryStage.setScene(new Scene(root, 500, 400));
    primaryStage.show();
  }
}
```

which produces



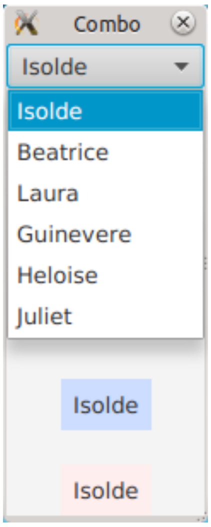
Combo-box:

Listing 167 JXV-Combo/Combo.java

```
package combo;

```

```java
import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.ComboBox;
import javafx.scene.control.Label;
import javafx.scene.layout.Priority;
import javafx.scene.layout.Region;
import javafx.scene.layout.VBox;
import javafx.scene.text.TextAlignment;
import javafx.stage.Stage;

public class Combo extends Application {
    public static void main(String[] args) {
        launch(args);
    }

    @Override public void start(Stage stage) {
        ComboBox<String> combo = new ComboBox<>();
        combo.getItems().addAll("Isolde", "Beatrice",
                                "Laura", "Guinevere",
                                "Heloise", "Juliet");

        Label lab1 = new Label();
        lab1.setTextAlignment(TextAlignment.CENTER);
        lab1.setStyle("-fx-background-color: #cdf;" +
                      "-fx-padding: 5pt;");
          // binding to selected item of the combobox
        lab1.textProperty().bind(combo.getSelectionModel()
                            .selectedItemProperty());

        Label lab2 = new Label();
        lab2.setTextAlignment(TextAlignment.CENTER);
        lab2.setStyle("-fx-background-color: #fee;" +
                      "-fx-padding: 5pt;");
          // or attach a listener to changes of the combobox
        combo.getSelectionModel().selectedItemProperty()
                                 .addListener(
            (observable, oldV, newV) ->
                lab2.setText(combo.getSelectionModel()
                                  .getSelectedItem())
        );

        combo.getSelectionModel().selectFirst();

          // just a vertical space...
        Region space = new Region();
        space.setPrefHeight(130);

        VBox root = new VBox(20);
        VBox.setVgrow(space, Priority.ALWAYS);
        root.setAlignment(Pos.CENTER);
        root.getChildren().addAll(combo, space, lab1, lab2);
        stage.setTitle("Combo");
        stage.setScene(new Scene(root));
        stage.show();
    }
}
```

which produces



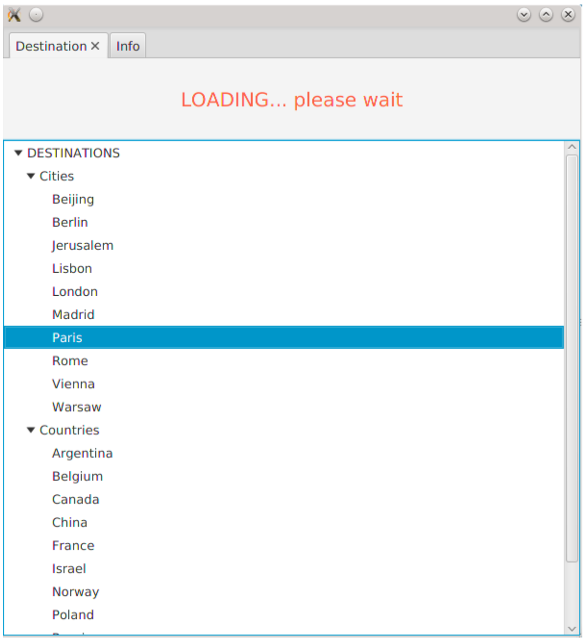
TabPane:

Listing 168 JXH-WebTabs/Tabs.java

```
package webtabs;

import java.util.Arrays;
import java.util.List;
import javafx.application.Application;
import javafx.beans.value.ChangeListener;
import javafx.beans.value.ObservableValue;
import javafx.concurrent.Worker.State;
import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.layout.Priority;
import javafx.scene.layout.VBox;
import javafx.scene.control.Label;
import javafx.scene.control.Tab;
import javafx.scene.control.TabPane;
import javafx.scene.control.TreeItem;
import javafx.scene.control.TreeView;
import javafx.scene.paint.Color;
import javafx.scene.text.Font;
import javafx.scene.web.WebEngine;
import javafx.scene.web.WebView;
```

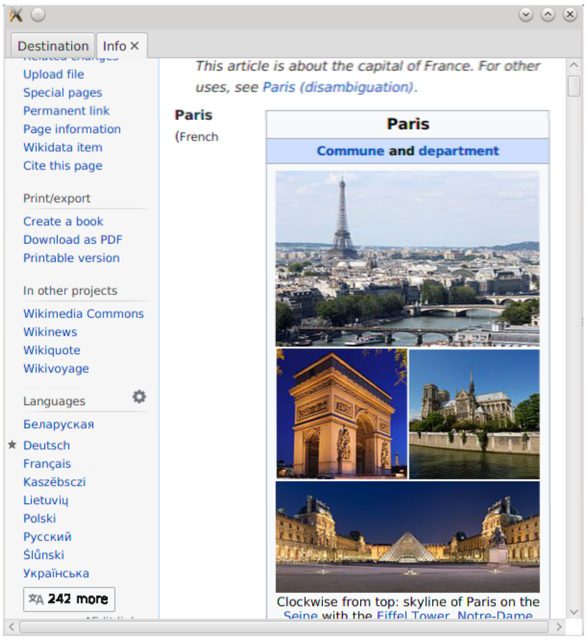
```java
import javafx.stage.Stage;

public class Tabs extends Application {

    public static void main(String[] args) {
        launch(args);
    }

    private static final List<String> cities =
        Arrays.asList(
            "Beijing", "Berlin", "Jerusalem", "Lisbon",
            "London", "Madrid", "Paris", "Rome",
            "Vienna", "Warsaw");
    private static final List<String> countries =
        Arrays.asList(
            "Argentina", "Belgium", "Canada", "China",
            "France", "Israel", "Norway", "Poland", "Russia",
            "Spain", "Ukraine");
    private static final String STR_WAIT =
            "LOADING... please wait";
    private static final String STR_SEL =
            "Select a destination below";
    private static final String WIKI =
            "https://en.wikipedia.org/wiki/";

    @Override
    public void start(Stage stage) {
        TabPane tabPane = new TabPane();
        Label label = new Label(STR_SEL);
        label.setFont(Font.font(20));
        label.setPadding(new Insets(30));
        label.setTextFill(Color.TOMATO);

        Tab tabWeb = new Tab("Info");
        WebView viewWeb = new WebView();
        WebEngine engWeb = viewWeb.getEngine();
        tabWeb.setContent(viewWeb);
        engWeb.getLoadWorker().stateProperty().addListener(
            new ChangeListener<State>() {
                public void changed(ObservableValue ob,
                            State oldV, State newV) {
                    if (newV == State.SUCCEEDED) {
                        label.setText(STR_SEL);
                        tabPane.getSelectionModel()
                               .select(tabWeb);
                    }
                }
            }
        );

        Tab tabTree = new Tab("Destination");
        TreeView<String> treeView = getTree();
        VBox grid = new VBox();
        grid.getChildren().addAll(label, treeView);
        VBox.setVgrow(treeView, Priority.ALWAYS);
        grid.setAlignment(Pos.TOP_CENTER);
        tabTree.setContent(grid);
        treeView
           .getSelectionModel()
           .selectedItemProperty()
           .addListener((obs, oldV, newV) -> {
                if(newV == null || !newV.isLeaf()) return;
                label.setText(STR_WAIT);
                engWeb.load(WIKI + newV.getValue());
            }
        );

        tabPane.getTabs().addAll(tabTree, tabWeb);

        stage.setScene(new Scene(tabPane));
        stage.show();
    }

    private static TreeView<String> getTree() {
        TreeItem<String> treeRoot =
                new TreeItem<>("DESTINATIONS");
        treeRoot.setExpanded(true);

        TreeItem<String> citiesNode =
                new TreeItem<>("Cities");
        for (String city : cities) {
            TreeItem<String> item = new TreeItem<>(city);
            citiesNode.getChildren().add(item);
        }
        treeRoot.getChildren().add(citiesNode);

        TreeItem<String> countriesNode =
                new TreeItem<>("Countries");
        for (String country : countries) {
            TreeItem<String> item = new TreeItem<>(country);
            countriesNode.getChildren().add(item);
        }
        treeRoot.getChildren().add(countriesNode);

        TreeView<String> treeView = new TreeView<>(treeRoot);
        return treeView;
    }
}
```

which produces



LOADING... please wait

```
▾ DESTINATIONS
  ▾ Cities
      Beijing
      Berlin
      Jerusalem
      Lisbon
      London
      Madrid
      Paris
      Rome
      Vienna
      Warsaw
  ▾ Countries
      Argentina
      Belgium
      Canada
      China
      France
      Israel
      Norway
      Poland
```



This article is about the capital of France. For other uses, see Paris (disambiguation).

Paris (French

Paris

Commune and department

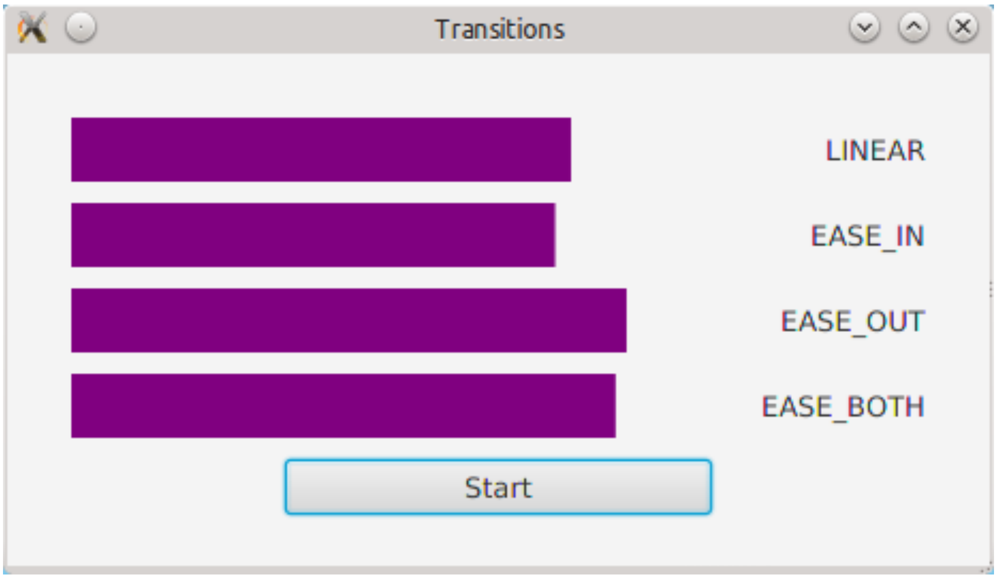
Clockwise from top: skyline of Paris on the Seine with the Eiffel Tower, Notre-Dame

Transitions:

Listing 169 JXQ-Transitions/Transitions.java

```java
package transitions;

import javafx.animation.Animation;
import javafx.animation.KeyFrame;
import javafx.animation.KeyValue;
import javafx.animation.Timeline;
import javafx.application.Application;
import javafx.geometry.HPos;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.layout.ColumnConstraints;
import javafx.scene.layout.GridPane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;
import javafx.util.Duration;
import static javafx.animation.Interpolator.*;

public class Transitions extends Application {
    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage stage) {
        double hG = 30, wD = 300, finalTime = 7;
        final int bars = 4;

        GridPane root = new GridPane();
        root.setPadding(new Insets(hG));
        root.setVgap(hG/3);
        root.getColumnConstraints().addAll(
                new ColumnConstraints(wD/3),
                new ColumnConstraints(wD/3),
                new ColumnConstraints(wD/3),
                new ColumnConstraints(wD/3));
        String[] inters = {
            "LINEAR", "EASE_IN", "EASE_OUT", "EASE_BOTH"};
        Rectangle[] rects = new Rectangle[bars];
        for (int i = 0; i < bars; ++i) {
            rects[i] = new Rectangle(0, hG);
            rects[i].setFill(Color.PURPLE);
            root.add(rects[i], 0, i, 3, 1);
            Label lab = new Label(inters[i]);
            root.add(lab, 3, i);
            GridPane.setHalignment(lab, HPos.RIGHT);
        }

        Timeline anim = new Timeline(
            new KeyFrame(Duration.seconds(finalTime),
                new KeyValue(rects[0].widthProperty(),
                                            wD, LINEAR),
                new KeyValue(rects[1].widthProperty(),
                                            wD, EASE_IN),
                new KeyValue(rects[2].widthProperty(),
                                            wD, EASE_OUT),
                new KeyValue(rects[3].widthProperty(),
                                            wD, EASE_BOTH)));
        anim.setCycleCount(Animation.INDEFINITE);
        Button start = new Button("Start");
        start.setMaxWidth(Double.MAX_VALUE);
        start.setPrefHeight(hG);
        start.setOnAction(e -> anim.playFromStart());
        root.add(start, 1, bars, 2, 1);
        GridPane.setHalignment(start, HPos.CENTER);
        Scene scene = new Scene(
                root, 4*wD/3 + 2*hG, (4*bars + 8)*hG/3);
        stage.setScene(scene);
        stage.setTitle("Transitions");
        stage.show();
    }
}
```

which produces

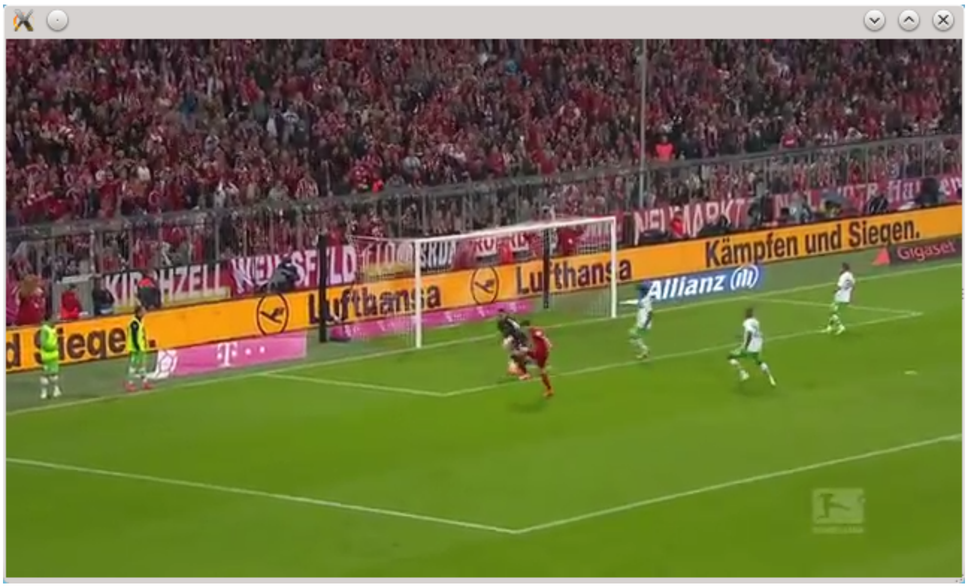


Media: basic example

Listing 170 JXW-BasicMedia/BasicPlayer.java

```java
package basicmedia;

// javafx.media must be included here

import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.media.Media;
import javafx.scene.media.MediaPlayer;
import javafx.scene.media.MediaView;
import javafx.stage.Stage;

import java.io.File;

public class BasicPlayer extends Application {
    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage stage) {
      // String sourceURL =
      //     "http://download.oracle.com/" +
      //     "otndocs/products/javafx/oow2010-2.flv";
      //  or
      // String sourceURL =
      //     "file:///home/werner/java/mediaplayer/rl.mp4";
      //  and then
      // Media media = new Media(sourceURL);
      //  or in one line (works also with jar archives)
        Media media = new Media(
                ClassLoader.getSystemResource("files/RL1.mp4")
                           .toExternalForm());
        MediaPlayer mplayer = new MediaPlayer(media);
        mplayer.setAutoPlay(true);
        MediaView mediaView = new MediaView(mplayer);
        mediaView.setFitWidth(640);
        mediaView.setFitHeight(360);
        mediaView.setPreserveRatio(true);
        Group root = new Group();
        root.getChildren().add(mediaView);
        stage.setScene(new Scene(root));
        stage.show();
    }
}
```

which shows the given video clip (with no controls, though. . .):

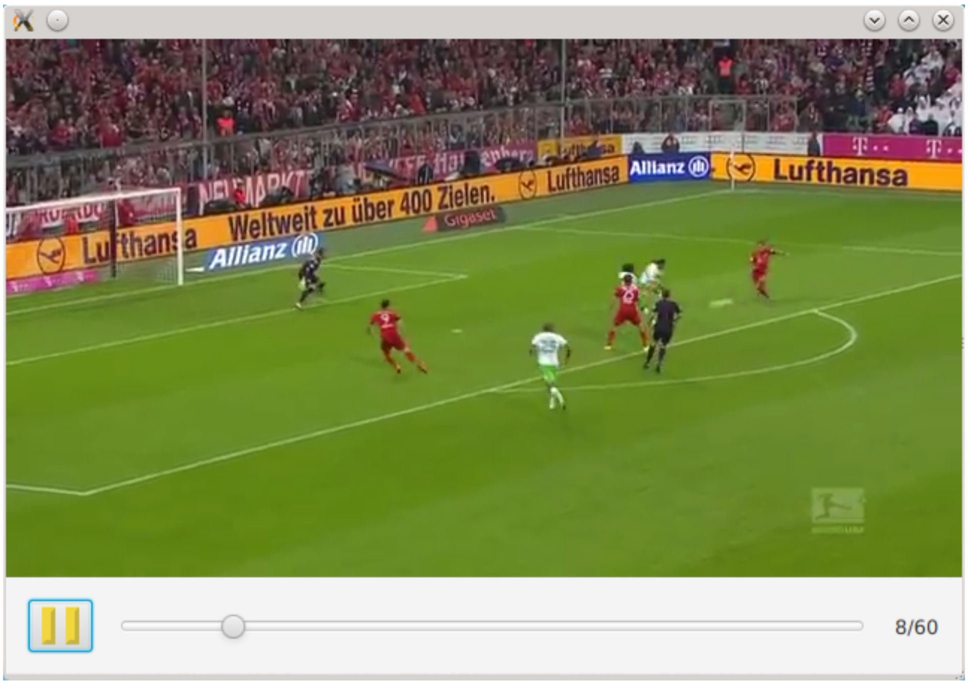


Media player with controls

Listing 171 JXX-DecentMedia/DecentPlayer.java

```java
package decentmedia;

// javafx.media must be added here

import javafx.application.Application;
import javafx.beans.InvalidationListener;
import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.Slider;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.HBox;
import javafx.scene.layout.Priority;
import javafx.scene.media.Media;
import javafx.scene.media.MediaPlayer;
import javafx.scene.media.MediaPlayer.Status;
import javafx.scene.media.MediaView;
import javafx.stage.Stage;
import javafx.util.Duration;

public class DecentPlayer extends Application {
    public static void main(String[] args) {
        launch(args);
    }

    private Duration duration;
    private double   durationSec;

    @Override
    public void start(Stage stage) {
        Media media = new Media(
            ClassLoader.getSystemResource("files/RL2.mp4")
                       .toExternalForm());
        MediaPlayer mpl = new MediaPlayer(media);
        mpl.setAutoPlay(false);
        MediaView mediaView = new MediaView(mpl);

        BorderPane root = new BorderPane();
        root.setCenter(mediaView);

        HBox controls = new HBox(15);
        controls.setAlignment(Pos.CENTER);
        controls.setPadding(new Insets(15));
        BorderPane.setAlignment(controls, Pos.CENTER);

          // icons
        ImageView pause = new ImageView(
            new Image(getClass()
                 .getResourceAsStream("/files/pause.png")));
        pause.setFitWidth(25); // height calculated to keep
        pause.setPreserveRatio(true); // the aspect ratio
        pause.setSmooth(true); // higher quality
        pause.setCache(true);  // improved performance
        ImageView play = new ImageView(
            new Image(getClass()
                 .getResourceAsStream("/files/play.png")));
        play.setFitWidth(25);
        play.setPreserveRatio(true);
        play.setSmooth(true);
        play.setCache(true);

          // play/pause button
        Button playPauseButton = new Button();
        playPauseButton.setGraphic(play);
        playPauseButton.setOnAction(e -> {
            Status status = mpl.getStatus();
            switch (status) {
                case UNKNOWN:  case HALTED:
                case DISPOSED: case STALLED:
                    return;
                case PAUSED: case READY: case STOPPED:
                    playPauseButton.setGraphic(pause);
                    mpl.play();
                    break;
                case PLAYING:
                    playPauseButton.setGraphic(play);
                    mpl.pause();
            }
        });
        controls.getChildren().add(playPauseButton);

          // progress slider
        Slider slider = new Slider();
        slider.setValue(0);
        slider.setMin(0); // max will be known after
                          // transition of mplayer to READY
           // slider will grow, but not button and label
        HBox.setHgrow(slider, Priority.ALWAYS);
        slider.setMinWidth(90);
        slider.setMaxWidth(Double.MAX_VALUE);
        InvalidationListener sliderInvalid = obs ->
            mpl.seek(Duration.seconds(slider.getValue()));
        slider.valueProperty().addListener(sliderInvalid);

        Label elapsedLab = new Label("0/00");

        mpl.setOnReady(() -> {
              // now we know duration and sizes of the media
              // so we can size the view and show the stage
            duration = mpl.getTotalDuration();
            durationSec = duration.toSeconds();
            slider.setMax(durationSec);
            elapsedLab.setText("0/" + (int)durationSec);
            mediaView.setFitWidth(media.getWidth());
            mediaView.setFitHeight(media.getHeight());
            mediaView.setPreserveRatio(true);
            stage.show();
        });

        mpl.setOnEndOfMedia(() -> {
              // rewind and pause
            playPauseButton.setGraphic(play);
            mpl.seek(mpl.getStartTime());
            mpl.pause();
            slider.setValue(0);
        });

          // InvalidationListener - we temporarily
          // disable slider's listener so the two
          // listeners do not 'compete'
        mpl.currentTimeProperty().addListener(obs -> {
              // InvalidationListener - we temporarily
              // disable slider's listener so the two
              // listeners do not 'compete'
            slider.valueProperty()
                  .removeListener(sliderInvalid);
            elapsedLab.setText((int)mpl.getCurrentTime()
                    .toSeconds() + "/" + (int)durationSec);
            slider.setValue(mpl.getCurrentTime()
                               .toSeconds());
            slider.valueProperty()
                  .addListener(sliderInvalid);
        });

        controls.getChildren().add(slider);
        controls.getChildren().add(elapsedLab);
        root.setBottom(controls);
        stage.setScene(new Scene(root));
          // stage will be shown in mpl.setOnReady,
          // when duration and sizes are known
    }
}
```

which shows the given video clip, this time with functional control bar:

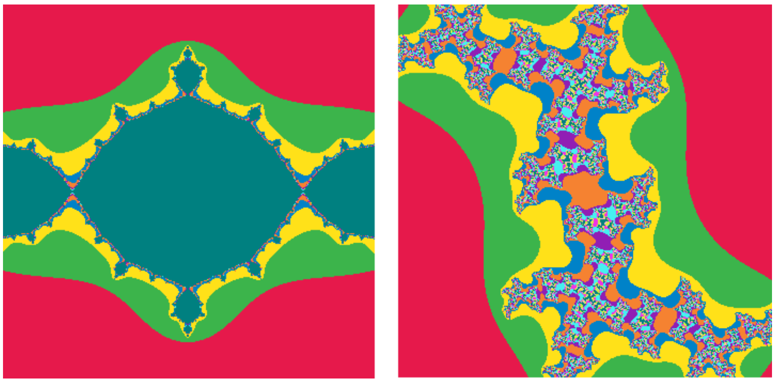


Setting individual pixels of the canvas using its graphics context and that graphics context's pixel writer:

Listing 172 JXY-CanvasPixels/Julia.java

```java
package julia;

// javafx.swing must be added here

import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;
import java.nio.ByteBuffer;
import javafx.application.Application;
import javafx.application.Platform;
import javafx.embed.swing.SwingFXUtils;
import javafx.scene.Scene;
import javafx.scene.canvas.Canvas;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.image.PixelWriter;
import javafx.scene.image.WritableImage;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.stage.Stage;
import javax.imageio.ImageIO;

public class Julia extends Application {
    public static void main(String[] args) {
        launch(args);
    }
      // 20 distinguishable colors + white and black
      // (by Sasha Trubetskoy - https://sashat.me)
    Color[] colors = {
            Color.web("#e6194b"), Color.web("#3cb44b"),
            Color.web("#ffe119"), Color.web("#0082c8"),
            Color.web("#f58231"), Color.web("#911eb4"),
            Color.web("#46f0f0"), Color.web("#f032e6"),
            Color.web("#d2f53c"), Color.web("#fabebe"),
            Color.web("#008080"), Color.web("#e6beff"),
            Color.web("#aa6e28"), Color.web("#fffac8"),
            Color.web("#800000"), Color.web("#aaffc3"),
            Color.web("#808000"), Color.web("#ffd8b1"),
            Color.web("#000080"), Color.web("#808080"),
            Color.web("#FFFFFF"), Color.web("#000000") };

    static final int NUM_COLS =  11, MAX_IT   = 35,
                     WIDTH    = 350, HEIGHT   = 350;
    @Override
    public void start(Stage stage) {
        if (NUM_COLS > colors.length) {
            System.err.println("NUM_COLS too big");
            Platform.exit();
        }
        Pane root = new Pane();
        Canvas canvas = new Canvas(WIDTH, HEIGHT);
        GraphicsContext gc = canvas.getGraphicsContext2D();

          // double cre = -1, cim = 0;
        double cre = 0.3, cim = 0.6;
        drawJuliaSet(gc, cre, cim);

        root.getChildren().add(canvas);
        stage.setScene(new Scene(root));
        stage.setTitle("Julia set: c = (" + cre +
                       ", " + cim + ")");
        stage.show();
    }

    private void drawJuliaSet(GraphicsContext gc,
                              double cre, double cim) {
        if (Math.max(Math.abs(cre), Math.abs(cim)) > 1.001) {
            System.out.println("c too big");
            Platform.exit();
        }
        PixelWriter pixelWriter = gc.getPixelWriter();
        final double ax =  2./(WIDTH-1),  bx = -1;
        final double ay = -2./(HEIGHT-1), by = +1;
        for(int py = 0; py < HEIGHT; ++py) {
            double zimT = ay*py + by;
            for(int px = 0; px < WIDTH; ++px) {
                double zreT = ax*px + bx;
                int it = MAX_IT;
                double zre = zreT, zim = zimT;
                for(int i = 1; i < MAX_IT && i < it; ++i) {
                    double ozre = zre, ozim = zim;
                    zre  = ozre*ozre - ozim*ozim + cre;
                    zim  = 2*ozre*ozim + cim;
                    if (zre*zre + zim*zim > 4) it = i;
                }
                pixelWriter.setColor(
                    px, py, colors[it*(NUM_COLS-1)/MAX_IT]);
            }
        }
          // saving canvas to a file
        try {
            WritableImage image =
                        new WritableImage(WIDTH, HEIGHT);
            gc.getCanvas().snapshot(null, image);
            BufferedImage bIm = SwingFXUtils
                               .fromFXImage(image, null);
            ImageIO.write(bIm, "png", new File("Julia.png"));
            System.out.println("File 'Julia.png' generated");
        } catch (IOException ex) {
            System.out.println("Error; file not generated");
        }
    }
}
```

For example, the figures below show two Julia sets for the square $\{-1, 1\} \times \{-1, 1\}$ of the complex plane with parameter $c = -1$ and $c = 0.3 + 0.6i$, respectively:



Embedding Swing components into a JavaFX application:

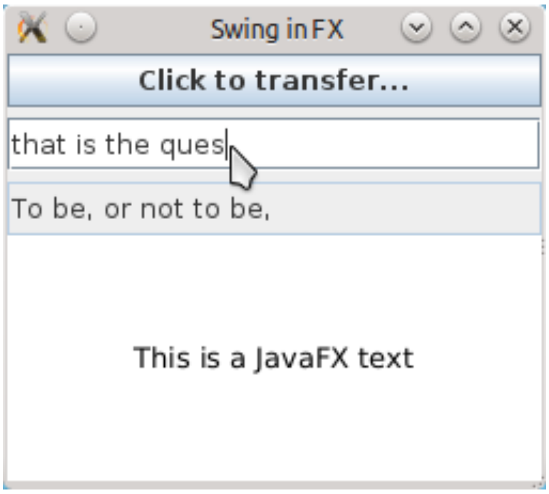
- Create a Swing component to be embedded into JavaFX GUI. Do not use heavy-weight components (like JFrame, but only light-weight ones; e.g., JPanel, which, of course, can contain other Swing light components as swingPanel in the example below;
- create an object of the SwingNode classi (variable swingNode in the example below);
- on the SwingNode, invoke setContent passing the Swing component to be embedded into JavaFX GUI; this must be executed on the Swing's event dispatch thread;

- add the **SwingNode** to the JavaFX GUI

Listing 173 JYJ-SwingInFX/SwingInFX.java

```
package swinginfx;

// javafx.swing must be added here


import java.awt.GridLayout;
import java.awt.event.ActionListener;
import javafx.application.Application;
import javafx.embed.swing.SwingNode;
import javafx.scene.Scene;
import javafx.scene.layout.VBox;
import javafx.scene.layout.StackPane;
import javafx.scene.text.Text;
import javafx.stage.Stage;

import javax.swing.JButton;
import javax.swing.JPanel;
import javax.swing.JTextField;
import javax.swing.SwingUtilities;

public class SwingInFX extends Application {
    public static void main(String[] args) {
        launch(args);
    }
    @Override
    public void start(Stage stage) {
          // swing part
        JButton click = new JButton("Click to transfer...");
        JTextField text = new JTextField(25);
        JTextField shadow = new JTextField(25);
        shadow.setEditable(false);
        ActionListener action = e -> {
            String s = text.getText();
            text.setText("");
            shadow.setText(s);
            text.requestFocus();
        };
        click.addActionListener(action);
        text.addActionListener(action);
        JPanel swingPanel =
            new JPanel(new GridLayout(3, 1, 5, 5));
        swingPanel.add(click);
        swingPanel.add(text);
        swingPanel.add(shadow);
          // embedding swing in SwingNode; setContent
          // must be on Swing's event dispatch thread!
        SwingNode swingNode = new SwingNode();
        SwingUtilities.invokeLater(new Runnable() {
            @Override
            public void run() {
                swingNode.setContent(swingPanel);
            }
        });
          // FX part
        StackPane fxPane = new StackPane();
        fxPane.setMinWidth(250);
        fxPane.setMaxWidth(999);
        fxPane.setPrefHeight(200);
        fxPane.getChildren()
              .add(new Text("This is a JavaFX text"));
          // putting everything together
        VBox vbox = new VBox();
        vbox.getChildren().addAll(swingNode, fxPane);

        stage.setTitle("Swing in FX");
        stage.setScene(new Scene(vbox));
        stage.show();
    }
}
```

The program displays a window (the main component being a **VBox**) containing two panes: the upper is a Swing **JPanel** embedded in a **SwingNode** and the lower one is a JavaFX **StackPane**.



that is the ques

To be, or not to be,

This is a JavaFX text

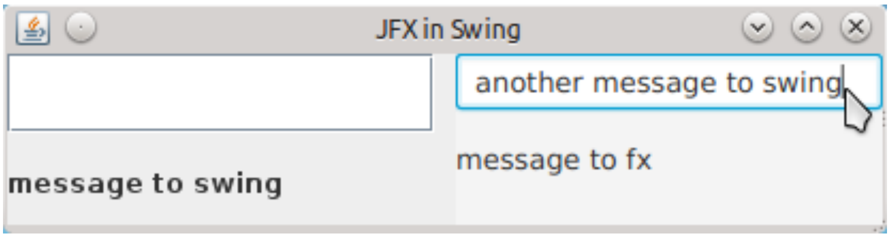
Embedding JFX components into a Swing application is illustrated below:

Listing 174 JYK-FXinSwing/FXinSwing.java

```
package fxinswing;

// javafx.swing must be added here

import java.awt.Dimension;
```

```java
import java.awt.GridLayout;
import javafx.application.Platform;
import javafx.embed.swing.JFXPanel;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.TilePane;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;
import javax.swing.SwingUtilities;
import static javafx.geometry.Orientation.VERTICAL;

public class FXinSwing {
    JTextField swingText;
    TextField  fxText;
    JLabel     swingLabel;
    Label      fxLabel;

    public static void main(String[] args) {
        new FXinSwing();
    }
    private FXinSwing() {
        swingText  = new JTextField(20);
        swingText.addActionListener(e -> {
            String s = swingText.getText();
            swingText.setText("");
              // must be on application thread,
              // as we touch a JFX component
            Platform.runLater(() -> fxLabel.setText(s));
        });
        swingLabel = new JLabel("Swing label");
        JPanel swingPanel =
                new JPanel(new GridLayout(2, 1, 0, 5));
        swingPanel.add(swingText);
        swingPanel.add(swingLabel);
        swingPanel.setPreferredSize(new Dimension(200,80));
        JFrame fr = new JFrame("JFX in Swing");
        fr.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        fr.setLayout(new GridLayout(1, 2, 10, 0));

        SwingUtilities.invokeLater(() -> {
            JFXPanel fx = new JFXPanel();
              // must be on application thread
            Platform.runLater(() -> fx.setScene(getFX()));
            fr.add(swingPanel);
            fr.add(fx);
            fr.pack();
            fr.setLocationRelativeTo(null);
            fr.setVisible(true);
        });
    }

    private Scene getFX() {
        TilePane root = new TilePane(VERTICAL, 0, 10);
        root.setPrefRows(2);
        fxText  = new TextField();
        fxText.setOnAction(a -> {
            String s = fxText.getText();
            fxText.setText("");
              // must be on EDT, as we
              // touch a swing component
            SwingUtilities.invokeLater(() ->
                swingLabel.setText(s)
            );
        });
        fxLabel = new Label("JFX label");
        fxText.setPrefWidth(200);
        fxLabel.setPrefWidth(200);
        root.setPrefSize(200,80);
        root.getChildren().addAll(fxText, fxLabel);
        return new Scene(root);
    }
}
```

The program displays a Swing JFrame containing two parts: Swing JPanel on the left and FX scene with a TilePane layout inside on the right. The FX scene is set on an object of type JFXPanel, which is a Swing 'wrapper' for FX components. Note that setting a listener for a Swing JTextField, we access the object fxLabel, which 'belongs' to JavaFX — therefore, we have to execute it on the application thread. In a similar way, when attaching a listener to FX TextField, we access swingLabel, so we have to do it on the EDT.



another message to swing

message to swing

message to fx

Listings

List of listings

Index